TURING

图灵程序设计丛书

大规模数据处理入门与实战

套装全10册





人民邮电出版社有限公司
POSTS & TELECOM PRESS Co.,LTD

VISIT...

LANZAROTE
Caliente.com

TURING

图灵程序设计丛书

大规模数据处理入门与实战

• 套装全10册 •





人民邮电出版社有限公司

POSTS & TELECOM PRESS Co.,LTD

总目录

Kafka权威指南
Flink基础教程
数据科学实战
SQL反模式
SQL必知必会（第4版）
Spark快速大数据分析
数据科学入门
Python数据挖掘入门与实践
Hadoop安全：大数据平台隐私保护
Hadoop数据分析

版权信息

书名：Kafka权威指南

作者：[美] Neha Narkhede Gwen Shapira Todd Palino

译者：薛命灯

ISBN：978-7-115-47327-1

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

091507240605ToBeReplacedWithUserId

版权声明
O'Reilly Media, Inc. 介绍
业界评论
序
前言
读者对象
排版约定
使用代码示例
O'Reilly Safari
联系我们
致谢
电子书
第 1 章　初识 Kafka
1.1　发布与订阅消息系统
1.1.1　如何开始
1.1.2　独立的队列系统
1.2　Kafka登场
1.2.1　消息和批次
1.2.2　模式
1.2.3　主题和分区
1.2.4　生产者和消费者
1.2.5　broker和集群
1.2.6　多集群
1.3　为什么选择Kafka
1.3.1　多个生产者
1.3.2　多个消费者
1.3.3　基于磁盘的数据存储
1.3.4　伸缩性
1.3.5　高性能
1.4　数据生态系统
使用场景
1.5　起源故事

1.5.1 LinkedIn的问题
1.5.2 Kafka的诞生
1.5.3 走向开源
1.5.4 命名
1.6 开始Kafka之旅
第 2 章 安装 Kafka
2.1 要事先行
2.1.1 选择操作系统
2.1.2 安装Java
2.1.3 安装Zookeeper
2.2 安装Kafka Broker
2.3 broker配置
2.3.1 常规配置
2.3.2 主题的默认配置
2.4 硬件的选择
2.4.1 磁盘吞吐量
2.4.2 磁盘容量
2.4.3 内存
2.4.4 网络
2.4.5 CPU
2.5 云端的Kafka
2.6 Kafka集群
2.6.1 需要多少个broker
2.6.2 broker配置
2.6.3 操作系统调优
2.7 生产环境的注意事项
2.7.1 垃圾回收器选项
2.7.2 数据中心布局
2.7.3 共享Zookeeper
2.8 总结
第 3 章 Kafka 生产者——向 Kafka 写入数据
3.1 生产者概览

3.2　创建Kafka生产者
3.3　发送消息到Kafka
3.3.1　同步发送消息
3.3.2　异步发送消息
3.4　生产者的配置
3.5　序列化器
3.5.1　自定义序列化器
3.5.2　使用Avro序列化
3.5.3　在Kafka里使用Avro
3.6　分区
3.7　旧版的生产者API
3.8　总结
第 4 章　Kafka 消费者——从 Kafka 读取数据
4.1　KafkaConsumer概念
4.1.1　消费者和消费者群组
4.1.2　消费者群组和分区再均衡
4.2　创建Kafka消费者
4.3　订阅主题
4.4　轮询
4.5　消费者的配置
4.6　提交和偏移量
4.6.1　自动提交
4.6.2　提交当前偏移量
4.6.3　异步提交
4.6.4　同步和异步组合提交
4.6.5　提交特定的偏移量
4.7　再均衡监听器
4.8　从特定偏移量处开始处理记录
4.9　如何退出
4.10　反序列化器
4.11　独立消费者——为什么以及怎样使用没有群组的消费者
4.12　旧版的消费者API

4.13　总结
第 5 章　深入 Kafka
5.1　集群成员关系
5.2　控制器
5.3　复制
5.4　处理请求
5.4.1　生产请求
5.4.2　获取请求
5.4.3　其他请求
5.5　物理存储
5.5.1　分区分配
5.5.2　文件管理
5.5.3　文件格式
5.5.4　索引
5.5.5　清理
5.5.6　清理的工作原理
5.5.7　被删除的事件
5.5.8　何时会清理主题
5.6　总结
第 6 章　可靠的数据传递
6.1　可靠性保证
6.2　复制
6.3　broker配置
6.3.1　复制系数
6.3.2　不完全的首领选举
6.3.3　最少同步副本
6.4　在可靠的系统里使用生产者
6.4.1　发送确认
6.4.2　配置生产者的重试参数
6.4.3　额外的错误处理
6.5　在可靠的系统里使用消费者
6.5.1　消费者的可靠性配置

6.5.2 显式提交偏移量
6.6 验证系统可靠性
6.6.1 配置验证
6.6.2 应用程序验证
6.6.3 在生产环境监控可靠性
6.7 总结
第 7 章 构建数据管道
7.1 构建数据管道时需要考虑的问题
7.1.1 及时性
7.1.2 可靠性
7.1.3 高吞吐量和动态吞吐量
7.1.4 数据格式
7.1.5 转换
7.1.6 安全性
7.1.7 故障处理能力
7.1.8 耦合性和灵活性
7.2 如何在Connect API和客户端API之间作出选择
7.3 Kafka Connect
7.3.1 运行Connect
7.3.2 连接器示例——文件数据源和文件数据池
7.3.3 连接器示例——从MySQL到ElasticSearch
7.3.4 深入理解Connect
7.4 Connect之外的选择
7.4.1 用于其他数据存储的摄入框架
7.4.2 基于图形界面的ETL工具
7.4.3 流式处理框架
7.5 总结
第 8 章 跨集群数据镜像
8.1 跨集群镜像的使用场景
8.2 多集群架构
8.2.1 跨数据中心通信的一些现实情况
8.2.2 Hub和Spoke架构

8.2.3　双活架构
8.2.4　主备架构
8.2.5　延展集群
8.3　KafkaのMirrorMaker
8.3.1　如何配置
8.3.2　在生产环境部署MirrorMaker
8.3.3　MirrorMaker调优
8.4　其他跨集群镜像方案
8.4.1　优步的uReplicator
8.4.2　Confluent的Replicator
8.5　总结
第 9 章　管理 Kafka
9.1　主题操作
9.1.1　创建主题
9.1.2　增加分区
9.1.3　删除主题
9.1.4　列出集群里的所有主题
9.1.5　列出主题详细信息
9.2　消费者群组
9.2.1　列出并描述群组
9.2.2　删除群组
9.2.3　偏移量管理
9.3　动态配置变更
9.3.1　覆盖主题的默认配置
9.3.2　覆盖客户端的默认配置
9.3.3　列出被覆盖的配置
9.3.4　移除被覆盖的配置
9.4　分区管理
9.4.1　首选的首领选举
9.4.2　修改分区副本
9.4.3　修改复制系数
9.4.4　转储日志片段

9.4.5　副本验证
9.5　消费和生产
9.5.1　控制台消费者
9.5.2　控制台生产者
9.6　客户端ACL
9.7　不安全的操作
9.7.1　移动集群控制器
9.7.2　取消分区重分配
9.7.3　移除待删除的主题
9.7.4　手动删除主题
9.8　总结
第 10 章　监控 Kafka
10.1　度量指标基础
10.1.1　度量指标在哪里
10.1.2　内部或外部度量
10.1.3　应用程序健康检测
10.1.4　度量指标的覆盖面
10.2　broker的度量指标
10.2.1　非同步分区
10.2.2　broker度量指标
10.2.3　主题和分区的度量指标
10.2.4　Java虚拟机监控
10.2.5　操作系统监控
10.2.6　日志
10.3　客户端监控
10.3.1　生产者度量指标
10.3.2　消费者度量指标
10.3.3　配额
10.4　延时监控
10.5　端到端监控
10.6　总结
第 11 章　流式处理

11.1 什么是流式处理
11.2 流式处理的一些概念
11.2.1 时间
11.2.2 状态
11.2.3 流和表的二元性
11.2.4 时间窗口
11.3 流式处理的设计模式
11.3.1 单个事件处理
11.3.2 使用本地状态
11.3.3 多阶段处理和重分区
11.3.4 使用外部查找——流和表的连接
11.3.5 流与流的连接
11.3.6 乱序的事件
11.3.7 重新处理
11.4 Streams示例
11.4.1 字数统计
11.4.2 股票市场统计
11.4.3 填充点击事件流
11.5 Kafka Streams的架构概览
11.5.1 构建拓扑
11.5.2 对拓扑进行伸缩
11.5.3 从故障中存活下来
11.6 流式处理使用场景
11.7 如何选择流式处理框架
11.8 总结
附录 A 在其他操作系统上安装 Kafka
A.1 在Windows上安装Kafka
A.1.1 使用Windows的Linux子系统
A.1.2 使用本地Java
A.2 在MacOS上安装Kafka
A.2.1 使用Homebrew
A.2.2 手动安装

作者介绍
封面介绍

版权声明

© 2017 by Neha Narkhede, Gwen Shapira, Todd Palino.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc.and Posts & Telecom Press, 2018. Authorized translation of the English edition, 2017 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2017。

简体中文版由人民邮电出版社出版，2018。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者—— O'Reilly Media,Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。 O'Reilly 的会议和

峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 **O'Reilly** 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 **Yogi Berra** 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，**Tim** 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

序

这是一个激动人心的时刻，成千上万的企业在使用 Kafka，三分之一多的世界 500 强公司也在其中。Kafka 是成长最快的开源项目之一，它的生态系统也在蓬勃发展。Kafka 正在成为管理和处理流式数据的利器。

Kafka 从何而来？我们为什么要开发 Kafka ？ Kafka 到底是什么？

Kafka 最初是 LinkedIn 的一个内部基础设施系统。我们发现，虽然有很多数据库和系统可以用来存储数据，但在我们的架构里，刚好缺一个可以帮助处理持续数据流的组件。在开发 Kafka 之前，我们实验了各种现成的解决方案，从消息系统到日志聚合系统，再到 ETL 工具，它们都无法满足我们的需求。

最后，我们决定从头开发一个系统。我们不想只是开发一个能够存储数据的系统，比如传统的关系型数据库、键值存储引擎、搜索引擎或缓存系统，我们希望能够把数据看成是持续变化和不断增长的流，并基于这样的想法构建出一个数据系统；事实上，是一个数据架构。

这个想法实现后比我们最初预想的适用性更广。Kafka 一开始被用在社交网络的实时应用和数据流当中，而现在已经成为下一代数据架构的基础。大型零售商正在基于持续数据流改造他们的基础业务流程，汽车公司正在从互联网汽车那里收集和处理实时数据流，银行也在重新思考基于 Kafka 改造他们的基础流程和系统。

那么 Kafka 在这当中充当了怎样的角色？它与现有的系统有什么区别？

我们认为 Kafka 是一个流平台：在这个平台上可以发布和订阅数据流，并把它们保存起来、进行处理，这就是构建 Kafka 的初衷。以这种方式来看待数据确实与人们习惯的想法有所不同，但它确实在构建应用和架构方面表现出了强大的抽象能力。Kafka 经常会被拿来与现有的技术作比较：企业级消息系统、大数据系统（如 Hadoop）和数据集成或 ETL 工具。这里的每一项比较都有一定的道理，但也有失偏颇。

Kafka 有点像消息系统，允许发布和订阅消息流。从这点来看，它类似于 ActiveMQ、 RabbitMQ 或 IBM 的 MQSeries 等产品。尽管看上去有些相似，但 Kafka 与这些传统的消息系统仍然存在很多重要的不同点，这些差异使它完全不同于消息系统。首先，作为一个现代的分布式系统，Kafka 以集群的方式运行，可以自由伸缩，处理公司的所有应用程序。Kafka 集群并不是一组独立运行的 broker，而是一个可以灵活伸缩的中心平台，可以处理整个公司所有的数据流。其次，Kafka 可以按照你的要求存储数据，保存多久都可以。作为数据连接层，Kafka 提供了数据传递保证——可复制、持久化，保留多长时间完全可以由你来决定。最后，流式处理将数据处理的层次提升到了新高度。消息系统只会传递消息，而 Kafka 的流式处理能力让你只用很

少的代码就能够动态地处理派生流和数据集。Kafka 的这些独到之处足以让你刮目相看，它不只是“另一个消息队列”。

从另一个角度来看 Kafka，我们会把它看成实时版的 Hadoop——这也是我们设计和构建 Kafka 的原始动机之一。Hadoop 可以存储和定期处理大量的数据文件，而 Kafka 可以存储和持续处理大型的数据流。从技术角度来看，它们有着惊人的相似之处，很多人将新兴的流式处理看成批处理的超集。它们之间的最大不同体现在持续的低延迟处理和批处理之间的差异上。Hadoop 和大数据主要应用在数据分析上，而 Kafka 因其低延迟的特点更适合用在核心的业务应用上。业务事件时刻在发生，Kafka 能够及时对这些事件作出响应，基于 Kafka 构建的服务直接为业务运营提供支撑，提升用户体验。

Kafka 与 ETL 工具或其他数据集成工具之间也可以进行一番比较。Kafka 和这些工具都擅长移动数据，但我想它们最大的不同在于 Kafka 颠覆了传统的思维。Kafka 并非只是把数据从一个系统拆解出来再塞进另一个系统，它其实是一个面向实时数据流的平台。也就是说，它不仅可以将现有的应用程序和数据系统连接起来，它还能用于加强这些触发相同数据流的应用。我们认为这种以数据流为中心的架构是非常重要的。在某种程度上说，这些数据流是现代数字科技公司的核心，与他们的现金流一样重要。

将上述的三个领域聚合在一起，将所有的数据流整合到一起，流平台因此变得极具吸引力。

当然，除了这些不同点之外，对于那些习惯了开发请求与响应风格应用和关系型数据库的人来说，要学会基于持续数据流构建应用程序也着实是一个巨大的思维转变。借助这本书来学习 Kafka 再好不过了，从内部架构到 API，都是由对 Kafka 最了解的人亲手呈现的。我希望你们能够像我一样喜欢这本书！

—— Jay Kreps，Confluent 联合创始人兼 CEO

前言

给予一个技术书籍作者最好的赞赏莫过于这句话——“如果在一开始接触这门技术时能看到这本书就好了”。在开始写这本书的时候，我们就是以这句话作为写作目标。我们开发 Kafka，在生产环境运行

Kafka，帮助很多公司构建基于 Kafka 的系统，帮助他们管理数据管道，积累了很多经验，但也困惑："应该把哪些东西分享给 Kafka 新用户，让他们从新手变成专家？"这本书就是我们日常工作最好的写照：运行 Kafka 并帮助其他人更好地使用 Kafka。

我们相信，书中提供的这些内容能够帮助 Kafka 用户在生产环境运行 Kafka 以及基于 Kafka 构建健壮的高性能应用程序。我们列举了一些非常流行的应用场景：用于事件驱动微服务系统的消息总线、流式应用和大规模数据管道。这本书通俗易懂，能够帮助每一个 Kafka 用户在任意的架构或应用场景里使用好 Kafka。书中介绍了如何安装和配置 Kafka、如何使用 Kafka API、Kafka 的设计原则和可靠性保证，以及 Kafka 的一些架构细节，如复制协议、控制器和存储层。我们相信，Kafka 的设计原理和内部架构不仅会成为分布式系统构建者的兴趣所在，对于那些在生产环境部署 Kafka 或使用 Kafka 构建应用程序的人来说也是非常有用的。越是了解 Kafka，就越是能够更好地作出权衡。

在软件工程里，条条道路通罗马，每一个问题都有多种解决方案。Kafka 为专家级别的用户提供了巨大的灵活性，而新手则需要克服陡峭的学习曲线才能成为专家。Kafka 通常会告诉你如何使用某个功能特性，但不会告诉你为什么要用它或者为什么不该用它。我们会尽可能地解释我们的设计决策和权衡背后的缘由，以及用户在哪些情况下应该或不应该使用 Kafka 提供的特性。

读者对象

这本书是为使用 Kafka API 开发应用程序的工程师和在生产环境安装、配置、调优、监控 Kafka 的运维工程师（也可以叫作 SRE、运维人员或系统管理员）而写的。我们也考虑到了数据架构师和数据工程师，他们负责设计和构建整个组织的数据基础架构。某些章节（特别是第 3 章、第 4 章和第 11 章）主要面向 Java 开发人员，并假设读者已经熟悉基本的 Java 语言编程，比如异常处理和并发编程。其他章节（特别是第 2 章、第 8 章、第 9 章和第 10 章）则假设读者在 Linux 的运行、存储和网络配置方面有一定的经验。本书的其余部分则讨论了一般性的软件架构，不要求读者具备特定的知识。

另一类可能对本书感兴趣的人是那些经理或架构师，他们不直接使用 Kafka，但会与使用 Kafka 的工程师打交道。他们有必要了解 Kafka 所能提供的保证机制，以及他们的同事在构建基于 Kafka 的系统时所作出的权衡。这本书可以成为企业管理人员的利器，确保他们的工程

师在 Kafka 方面训练有素，让他们的团队了解他们本该知道的知识。

排版约定

本书使用了下列排版约定。

- 黑体

  表示新术语或重点强调的内容。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）

  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`constant width italic`*）

  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注记。



该图标表示警告或警示。

使用代码示例

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许

可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。例如“Kafka 权威指南，作者 Neha Narkhede、Gwen Shapira 和 Todd Palino（O'Reilly），版权归 Neha Narkhede、Gwen Shapira 和 Todd Palino 所有，978-1-4919-3616-0”。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@ oreilly.com 与我们联系。

O'Reilly Safari



Safari（原来叫 Safari Books Online）是面向企业、政府、教育从业者和个人的会员制培训和参考咨询平台。

我们向会员开放成千上万本图书以及培训视频、学习路线、交互式教程和专业视频。这些资源来自 250 多家出版机构，其中包括 O'Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett 和 Course Technology。

更多信息，请访问 http://oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是 http://oreil.ly/2tVmYjk。

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

我们想感谢众多为 Kafka 和它的生态系统做出贡献的人。如果没有他们艰辛的工作，就不会有这本书的问世。特别感谢 Jay Kreps、Neha Narkhede 和 Jun Rao，以及他们在 LinkedIn 的同事和领导，他们创造了 Kafka，并把它捐献给了 Apache 软件基金会。

很多人在早前为本书提供了很多有价值的反馈，我们非常感激他们为此付出的时间，也很钦佩他们的专业能力，这些人包括：Apurva Mehta、Arseniy Tashoyan、Dylan Scott、Ewen Cheslack-Postava、Grant Henke、Ismael Juma、James Cheng、Jason Gustafson、Jeff

Holoman、Joel Koshy、Jonathan Seidman、Matthias Sax、Michael Noll、Paolo Castagna。我们还想感谢众多在网站上留下评论和反馈的读者。

很多审稿人提供了有价值的意见，极大改进了本书的质量。书中的遗留错误理应由我们作者负责。

我们要感谢 O'Reilly 编辑 Shannon Cutt 的鼓励、耐心和深谋远虑。对于一个作者来说，与 O'Reilly 一起合作是一段非凡的经历——他们所提供的支持，从工具到签名售书，都是无可匹敌的。我们感谢每一个参与本书相关工作的人，很感激他们愿意与我们一起工作。

另外，我们也想感谢我们的领导和同事，感谢他们在我们写作这本书的过程中给予的帮助和鼓励。

Gwen 要感谢她的丈夫 Omer Shapira，在她写书的几个月时间里，他一直给予她支持和耐心。还有她的父亲 Lior Shapira，让她学会了如何在困难面前不轻言放弃，尽管这种生活哲学总是让她麻烦不断。

Todd 要感谢他的妻子 Marcy 和女儿 Bella 及 Kaylee，她们一直在背后默默地支持他。因为有了她们的支持，他才有更多的时间写作，才能厘清思路，坚持到最后。

电子书

扫描如下二维码，即可购买本书电子版。



第 1 章　初识 Kafka

数据为企业的发展提供动力。我们从数据中获取信息，对它们进行分析处理，然后生成更多的数据。每个应用程序都会产生数据，包括日志消息、度量指标、用户活动记录、响应消息等。数据的点点滴滴都在暗示一些重要的事情，比如下一步行动的方向。我们把数据从源头移动到可以对它们进行分析处理的地方，然后把得到的结果应用到实际场景中，这样才能够确切地知道这些数据要告诉我们什么。例如，我们每天在 Amazon 网站上浏览感兴趣的商品，浏览信息被转化成商品推荐，并在稍后展示给我们。

这个过程完成得越快，组织的反应就越敏捷。花费越少的精力在数据移动上，就越能专注于核心业务。这就是为什么在一个以数据为驱动的企业里，数据管道会成为关键性组件。如何移动数据，几乎变得与数据本身一样重要。

> 每一次科学家们发生分歧，都是因为掌握的数据不够充分。所以我们可以先就获取哪一类数据达成一致。只要获取了数据，问题也就迎刃而解了。要么我是对的，要么你是对的，要么我们都是错的。然后我们继续研究。

——Neil deGrasse Tyson

1.1　发布与订阅消息系统

在正式讨论 Apache Kafka（以下简称 Kafka）之前，先来了解发布与订阅消息系统的概念，并认识这个系统的重要性。数据（消息）的发送者（发布者）不会直接把消息发送给接收者，这是发布与订阅消息系统的一个特点。发布者以某种方式对消息进行分类，接收者（订阅者）订阅它们，以便接收特定类型的消息。发布与订阅系统一般会有一个 broker，也就是发布消息的中心点。

1.1.1　如何开始

发布与订阅消息系统的大部分应用场景都是从一个简单的消息队列或一个进程间通道开始的。例如，你的应用程序需要往别处发送监控信息，可以直接在你的应用程序和另一个可以在仪表盘上显示度量指标的应用程序之间建立连接，然后通过这个连接推送度量指标，如图 1-1 所示。

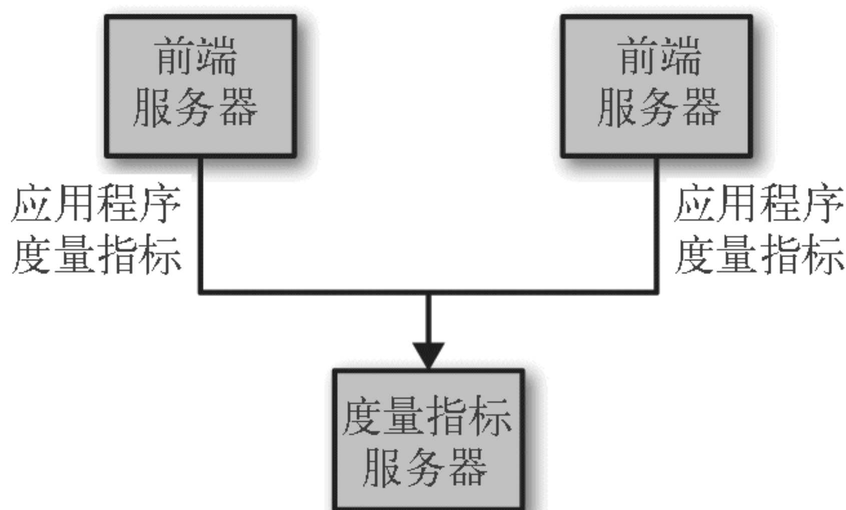


图 **1-1**：单个直连的度量指标发布者

这是刚接触监控系统时简单问题的应对方案。过了不久，你需要分析更长时间片段的度量指标，而此时的仪表盘程序满足不了需求，于是，你启动了一个新的服务来接收度量指标。该服务把度量指标保存

起来，然后进行分析。与此同时，你修改了原来的应用程序，把度量指标同时发送到两个仪表盘系统上。现在，你又多了 3 个可以生成度量指标的应用程序，它们都与这两个服务直接相连。而你的同事认为最好可以对这些服务进行轮询以便获得告警功能，于是你为每一个应用程序增加了一个服务器，用于提供度量指标。再过一阵子，有更多的应用程序出于各自的目的，都从这些服务器获取度量指标。这时的架构看起来就像图 1-2 所示的那样，节点间的连接一团糟。

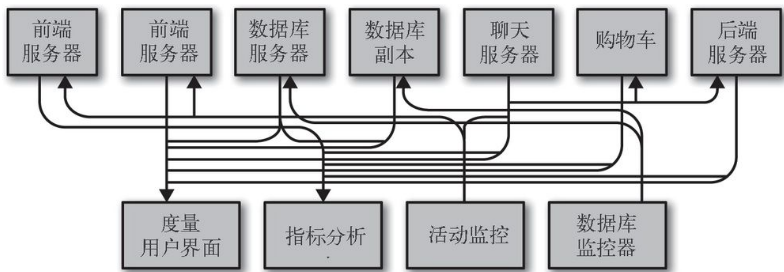


图 **1-2**：多个直连的度量指标发布者

这时，技术债务开始凸显出来，于是你决定偿还掉一些。你创建了一个独立的应用程序，用于接收来自其他应用程序的度量指标，并为其他系统提供了一个查询服务器。这样，之前架构的复杂度被降低到图 1-3 所示的那样。那么恭喜你，你已经创建了一个基于发布与订阅的消息系统。

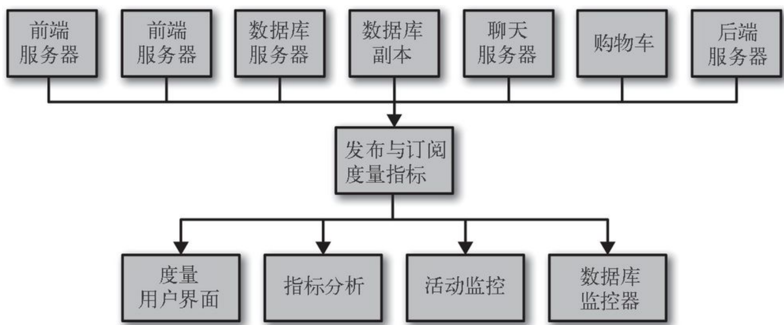


图 **1-3**：度量指标发布与订阅系统

1.1.2　独立的队列系统

在你跟度量指标打得不可开交的时候，你的一个同事也正在跟日志消息奋战。还有另一个同事正在跟踪网站用户的行为，为负责机器学习

开发的同事提供信息，同时为管理团队生成报告。你和同事们使用相同的方式创建这些系统，解耦信息的发布者和订阅者。图 1-4 所示的架构包含了 3 个独立的发布与订阅系统。

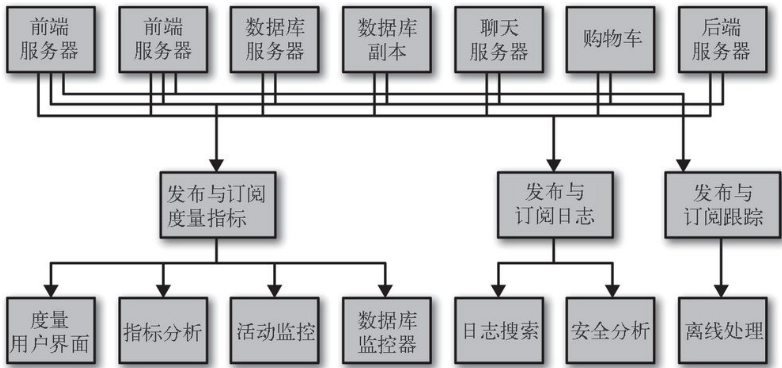


图 **1-4**：多个发布与订阅系统

这种方式比直接使用点对点的连接（图 1-2）要好得多，但这里有太多重复的地方。你的公司因此要为数据队列维护多个系统，每个系统又有各自的缺陷和不足。而且，接下来可能会有更多的场景需要用到消息系统。此时，你真正需要的是一个单一的集中式系统，它可以用来发布通用类型的数据，其规模可以随着公司业务的增长而增长。

1.2 Kafka登场

Kafka 就是为了解决上述问题而设计的一款基于发布与订阅的消息系统。它一般被称为“分布式提交日志”或者“分布式流平台”。文件系统或数据库提交日志用来提供所有事务的持久记录，通过重放这些日志可以重建系统的状态。同样地，Kafka 的数据是按照一定顺序持久化保存的，可以按需读取。此外，Kafka 的数据分布在整个系统里，具备数据故障保护和性能伸缩能力。

1.2.1 消息和批次

Kafka 的数据单元被称为消息。如果你在使用 Kafka 之前已经有数据库使用经验，那么可以把消息看成是数据库里的一个“数据行”或一条“记录”。消息由字节数组组成，所以对于 Kafka 来说，消息里的数据没有特别的格式或含义。消息可以有一个可选的元数据，也就是键。键也是一个字节数组，与消息一样，对于 Kafka 来说也没有特殊

的含义。当消息以一种可控的方式写入不同的分区时，会用到键。最简单的例子就是为键生成一个一致性散列值，然后使用散列值对主题分区数进行取模，为消息选取分区。这样可以保证具有相同键的消息总是被写到相同的分区上。第 3 章将详细介绍键的用法。

为了提高效率，消息被分批次写入 Kafka。批次就是一组消息，这些消息属于同一个主题和分区。如果每一个消息都单独穿行于网络，会导致大量的网络开销，把消息分成批次传输可以减少网络开销。不过，这要在时间延迟和吞吐量之间作出权衡：批次越大，单位时间内处理的消息就越多，单个消息的传输时间就越长。批次数据会被压缩，这样可以提升数据的传输和存储能力，但要做更多的计算处理。

1.2.2　模式

对于 Kafka 来说，消息不过是晦涩难懂的字节数组，所以有人建议用一些额外的结构来定义消息内容，让它们更易于理解。根据应用程序的需求，消息模式（schema）有许多可用的选项。像 JSON 和 XML 这些简单的系统，不仅易用，而且可读性好。不过，它们缺乏强类型处理能力，不同版本之间的兼容性也不是很好。Kafka 的许多开发者喜欢使用 Apache Avro，它最初是为 Hadoop 开发的一款序列化框架。Avro 提供了一种紧凑的序列化格式，模式和消息体是分开的，当模式发生变化时，不需要重新生成代码；它还支持强类型和模式进化，其版本既向前兼容，也向后兼容。

数据格式的一致性对于 Kafka 来说很重要，它消除了消息读写操作之间的耦合性。如果读写操作紧密地耦合在一起，消息订阅者需要升级应用程序才能同时处理新旧两种数据格式。在消息订阅者升级了之后，消息发布者才能跟着升级，以便使用新的数据格式。新的应用程序如果需要使用数据，就要与消息发布者发生耦合，导致开发者需要做很多繁杂的工作。定义良好的模式，并把它们存放在公共仓库，可以方便我们理解 Kafka 的消息结构。第 3 章将详细讨论模式和序列化。

1.2.3　主题和分区

Kafka 的消息通过主题进行分类。主题就好比数据库的表，或者文件系统里的文件夹。主题可以被分为若干个分区，一个分区就是一个提交日志。消息以追加的方式写入分区，然后以先入先出的顺序读取。要注意，由于一个主题一般包含几个分区，因此无法在整个主题范围内保证消息的顺序，但可以保证消息在单个分区内的顺序。图 1-5 所

示的主题有 4 个分区，消息被追加写入每个分区的尾部。Kafka 通过分区来实现数据冗余和伸缩性。分区可以分布在不同的服务器上，也就是说，一个主题可以横跨多个服务器，以此来提供比单个服务器更强大的性能。

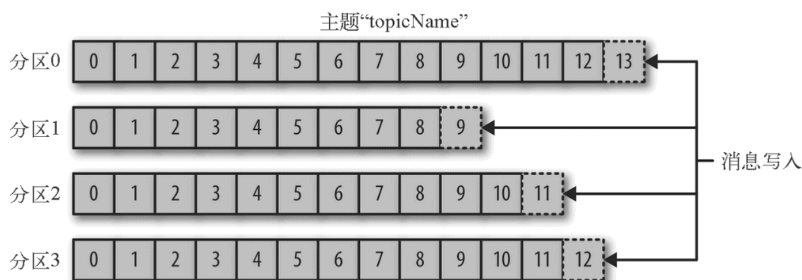


图 **1-5**：包含多个分区的主题表示

我们通常会使用流这个词来描述 Kafka 这类系统的数据。很多时候，人们把一个主题的数据看成一个流，不管它有多少个分区。流是一组从生产者移动到消费者的数据。当我们讨论流式处理时，一般都是这样描述消息的。Kafka Streams、Apache Samza 和 Storm 这些框架以实时的方式处理消息，也就是所谓的流式处理。我们可以将流式处理与离线处理进行比较，比如 Hadoop 就是被设计用于在稍后某个时刻处理大量的数据。第 11 章将会介绍流式处理。

1.2.4 生产者和消费者

Kafka 的客户端就是 Kafka 系统的用户，它们被分为两种基本类型：生产者和消费者。除此之外，还有其他高级客户端 API——用于数据集成的 Kafka Connect API 和用于流式处理的 Kafka Streams。这些高级客户端 API 使用生产者和消费者作为内部组件，提供了高级的功能。

生产者创建消息。在其他发布与订阅系统中，生产者可能被称为发布者或写入者。一般情况下，一个消息会被发布到一个特定的主题上。生产者在默认情况下把消息均衡地分布到主题的所有分区上，而并不关心特定消息会被写到哪个分区。不过，在某些情况下，生产者会把消息直接写到指定的分区。这通常是通过消息键和分区器来实现的，分区器为键生成一个散列值，并将其映射到指定的分区上。这样可以保证包含同一个键的消息会被写到同一个分区上。生产者也可以使用自定义的分区器，根据不同的业务规则将消息映射到分区。第 3 章将详细介绍生产者。

消费者读取消息。在其他发布与订阅系统中，消费者可能被称为订阅者或读者。消费者订阅一个或多个主题，并按照消息生成的顺序读取它们。消费者通过检查消息的偏移量来区分已经读取过的消息。偏移量是另一种元数据，它是一个不断递增的整数值，在创建消息时，Kafka 会把它添加到消息里。在给定的分区里，每个消息的偏移量都是唯一的。消费者把每个分区最后读取的消息偏移量保存在 Zookeeper 或 Kafka 上，如果消费者关闭或重启，它的读取状态不会丢失。

消费者是消费者群组的一部分，也就是说，会有一个或多个消费者共同读取一个主题。群组保证每个分区只能被一个消费者使用。图 1-6 所示的群组中，有 3 个消费者同时读取一个主题。其中的两个消费者各自读取一个分区，另外一个消费者读取其他两个分区。消费者与分区之间的映射通常被称为消费者对分区的所有权关系。

通过这种方式，消费者可以消费包含大量消息的主题。而且，如果一个消费者失效，群组里的其他消费者可以接管失效消费者的工作。第 4 章将详细介绍消费者和消费者群组。

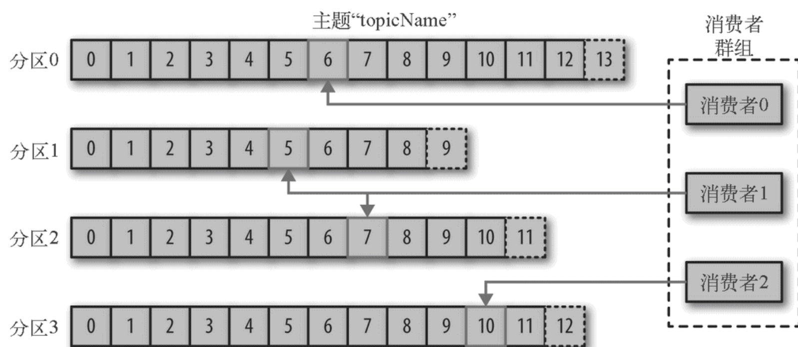


图 **1-6**：消费者群组从主题读取消息

1.2.5　broker和集群

一个独立的 Kafka 服务器被称为 **broker**。broker 接收来自生产者的消息，为消息设置偏移量，并提交消息到磁盘保存。broker 为消费者提供服务，对读取分区的请求作出响应，返回已经提交到磁盘上的消息。根据特定的硬件及其性能特征，单个 broker 可以轻松处理数千个分区以及每秒百万级的消息量。

broker 是集群的组成部分。每个集群都有一个 broker 同时充当了集

群控制器的角色（自动从集群的活跃成员中选举出来）。控制器负责管理工作，包括将分区分配给 broker 和监控 broker。在集群中，一个分区从属于一个 broker，该 broker 被称为分区的首领。一个分区可以分配给多个 broker，这个时候会发生分区复制（见图 1-7）。这种复制机制为分区提供了消息冗余，如果有一个 broker 失效，其他 broker 可以接管领导权。不过，相关的消费者和生产者都要重新连接到新的首领。第 6 章将详细介绍集群的操作，包括分区复制。

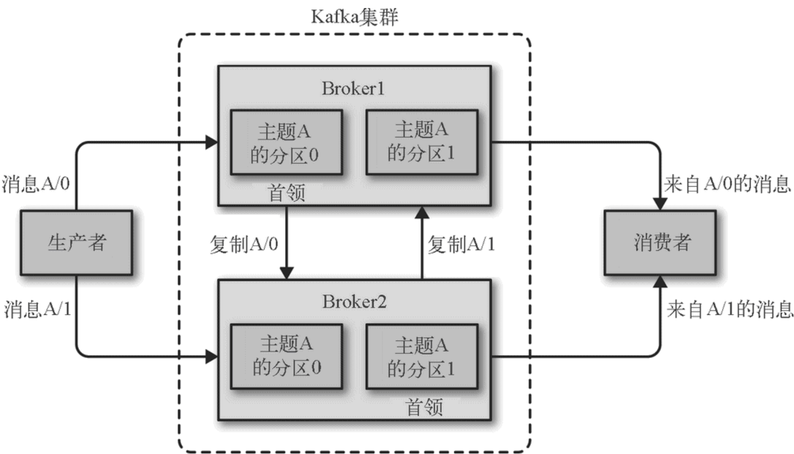


图 **1-7**：集群里的分区复制

保留消息（在一定期限内）是 Kafka 的一个重要特性。Kafka broker 默认的消息保留策略是这样的：要么保留一段时间（比如 7 天），要么保留到消息达到一定大小的字节数（比如 1GB）。当消息数量达到这些上限时，旧消息就会过期并被删除，所以在任何时刻，可用消息的总量都不会超过配置参数所指定的大小。主题可以配置自己的保留策略，可以将消息保留到不再使用它们为止。例如，用于跟踪用户活动的数据可能需要保留几天，而应用程序的度量指标可能只需要保留几个小时。可以通过配置把主题当作紧凑型日志，只有最后一个带有特定键的消息会被保留下来。这种情况对于变更日志类型的数据来说比较适用，因为人们只关心最后时刻发生的那个变更。

1.2.6　多集群

随着 Kafka 部署数量的增加，基于以下几点原因，最好使用多个集群。

- 数据类型分离

- 安全需求隔离
- 多数据中心（灾难恢复）

如果使用多个数据中心，就需要在它们之间复制消息。这样，在线应用程序才可以访问到多个站点的用户活动信息。例如，如果一个用户修改了他们的资料信息，不管从哪个数据中心都应该能看到这些改动。或者多个站点的监控数据可以被聚集到一个部署了分析程序和告警系统的中心位置。不过，Kafka 的消息复制机制只能在单个集群里进行，不能在多个集群之间进行。

Kafka 提供了一个叫作 **MirrorMaker** 的工具，可以用它来实现集群间的消息复制。MirrorMaker 的核心组件包含了一个生产者和一个消费者，两者之间通过一个队列相连。

消费者从一个集群读取消息，生产者把消息发送到另一个集群上。图 1-8 展示了一个使用 MirrorMaker 的例子，两个“本地”集群的消息被聚集到一个“聚合”集群上，然后将该集群复制到其他数据中心。不过，这种方式在创建复杂的数据管道方面显得有点力不从心。第 7 章将详细讨论这些案例。

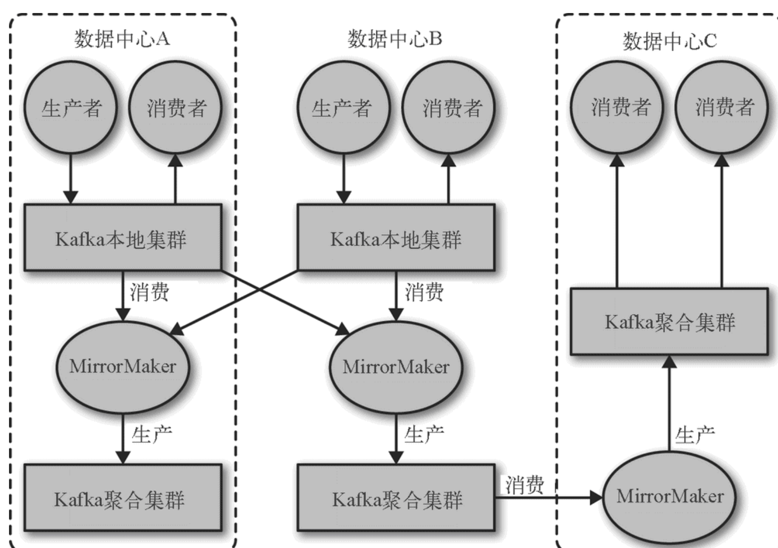


图 **1-8**：多数据中心架构

1.3 为什么选择Kafka

基于发布与订阅的消息系统那么多，为什么 Kafka 会是一个更好的选择呢？

1.3.1　多个生产者

Kafka 可以无缝地支持多个生产者，不管客户端在使用单个主题还是多个主题。所以它很适合用来从多个前端系统收集数据，并以统一的格式对外提供数据。例如，一个包含了多个微服务的网站，可以为页面视图创建一个单独的主题，所有服务都以相同的消息格式向该主题写入数据。消费者应用程序会获得统一的页面视图，而无需协调来自不同生产者的数据流。

1.3.2　多个消费者

除了支持多个生产者外，Kafka 也支持多个消费者从一个单独的消息流上读取数据，而且消费者之间互不影响。这与其他队列系统不同，其他队列系统的消息一旦被一个客户端读取，其他客户端就无法再读取它。另外，多个消费者可以组成一个群组，它们共享一个消息流，并保证整个群组对每个给定的消息只处理一次。

1.3.3　基于磁盘的数据存储

Kafka 不仅支持多个消费者，还允许消费者非实时地读取消息，这要归功于 Kafka 的数据保留特性。消息被提交到磁盘，根据设置的保留规则进行保存。每个主题可以设置单独的保留规则，以便满足不同消费者的需求，各个主题可以保留不同数量的消息。消费者可能会因为处理速度慢或突发的流量高峰导致无法及时读取消息，而持久化数据可以保证数据不会丢失。消费者可以在进行应用程序维护时离线一小段时间，而无需担心消息丢失或堵塞在生产者端。消费者可以被关闭，但消息会继续保留在 Kafka 里。消费者可以从上次中断的地方继续处理消息。

1.3.4　伸缩性

为了能够轻松处理大量数据，Kafka 从一开始就被设计成一个具有灵活伸缩性的系统。用户在开发阶段可以先使用单个 broker，再扩展到包含 3 个 broker 的小型开发集群，然后随着数据量不断增长，部署到生产环境的集群可能包含上百个 broker。对在线集群进行扩展丝毫不影响整体系统的可用性。也就是说，一个包含多个 broker 的集群，即使个别 broker 失效，仍然可以持续地为客户提供服务。要提

高集群的容错能力，需要配置较高的复制系数。第 6 章将讨论关于复制的更多细节。

1.3.5 高性能

上面提到的所有特性，让 Kafka 成为了一个高性能的发布与订阅消息系统。通过横向扩展生产者、消费者和 broker，Kafka 可以轻松处理巨大的消息流。在处理大量数据的同时，它还能保证亚秒级的消息延迟。

1.4 数据生态系统

已经有很多应用程序加入到了数据处理的大军中。我们定义了输入和应用程序，负责生成数据或者把数据引入系统。我们定义了输出，它们可以是度量指标、报告或者其他类型的数据。我们创建了一些循环，使用一些组件从系统读取数据，对读取的数据进行处理，然后把它们导到数据基础设施上，以备不时之需。数据类型可以多种多样，每一种数据类型可以有不同的内容、大小和用途。

Kafka 为数据生态系统带来了循环系统，如图 1-9 所示。它在基础设施的各个组件之间传递消息，为所有客户端提供一致的接口。当与提供消息模式的系统集成时，生产者和消费者之间不再有紧密的耦合，也不需要在它们之间建立任何类型的直连。我们可以根据业务需要添加或移除组件，因为生产者不再关心谁在使用数据，也不关心有多少个消费者。

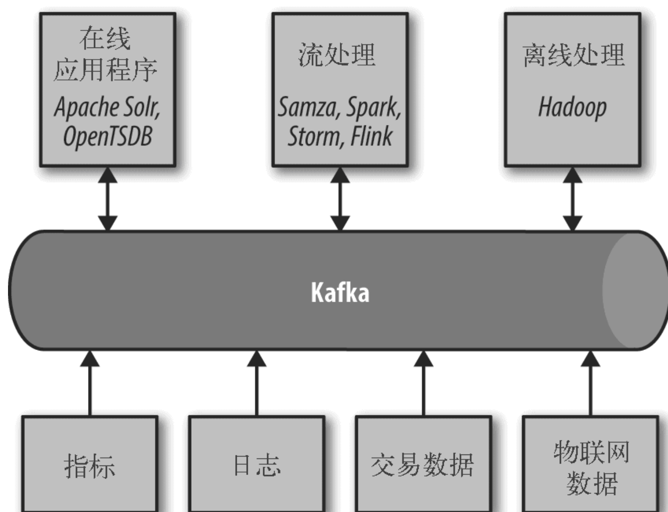


图 **1-9**：大数据生态系统

使用场景

01. 活动跟踪

Kafka 最初的使用场景是跟踪用户的活动。网站用户与前端应用程序发生交互，前端应用程序生成用户活动相关的消息。这些消息可以是一些静态的信息，比如页面访问次数和点击量，也可以是一些复杂的操作，比如添加用户资料。这些消息被发布到一个或多个主题上，由后端应用程序负责读取。这样，我们就可以生成报告，为机器学习系统提供数据，更新搜索结果，或者实现其他更多的功能。

02. 传递消息

Kafka 的另一个基本用途是传递消息。应用程序向用户发送通知（比如邮件）就是通过传递消息来实现的。这些应用程序组件可以生成消息，而不需要关心消息的格式，也不需要关心消息是如何被发送的。一个公共应用程序会读取这些消息，对它们进行处理：

- 格式化消息（也就是所谓的装饰）；
- 将多个消息放在同一个通知里发送；
- 根据用户配置的首选项来发送数据。

使用公共组件的好处在于，不需要在多个应用程序上开发重复的功能，而且可以在公共组件上做一些有趣的转换，比如把多个消息聚合成一个单独的通知，而这些工作是无法在其他地方完成的。

06. 度量指标和日志记录

Kafka 也可以用于收集应用程序和系统度量指标以及日志。Kafka 支持多个生产者的特性在这个时候就可以派上用场。应用程序定期把度量指标发布到 Kafka 主题上，监控系统或告警系统读取这些消息。Kafka 也可以用在像 Hadoop 这样的离线系统上，进行较长时间片段的数据分析，比如年度增长走势预测。日志消息也可以被发布到 Kafka 主题上，然后被路由到专门的日志搜索系统（比如 Elasticsearch）或安全分析应用程序。更改目标系统（比如日志存储系统）不会影响到前端应用或聚合方法，这是 Kafka 的另一个优点。

07. 提交日志

Kafka 的基本概念来源于提交日志，所以使用 Kafka 作为提交日志是件顺理成章的事。我们可以把数据库的更新发布到 Kafka 上，应用程序通过监控事件流来接收数据库的实时更新。这种变更日志流也可以用于把数据库的更新复制到远程系统上，或者合并多个应用程序的更新到一个单独的数据库视图上。数据持久化为变更日志提供了缓冲区，也就是说，如果消费者应用程序发生故障，可以通过重放这些日志来恢复系统状态。另外，紧凑型日志主题只为每个键保留一个变更数据，所以可以长时间使用，不需要担心消息过期问题。

08. 流处理

流处理是又一个能提供多种类型应用程序的领域。可以说，它们提供的功能与 Hadoop 里的 map 和 reduce 有点类似，只不过它们操作的是实时数据流，而 Hadoop 则处理更长时间片段的数据，可能是几个小时或者几天，Hadoop 会对这些数据进行批处理。通过使用流式处理框架，用户可以编写小型应用程序来操作 Kafka 消息，比如计算度量指标，为其他应用程序有效地处理消息分区，或者对来自多个数据源的消息进行转换。第 11 章将通过其他案例介绍流处理。

1.5 起源故事

Kafka 是为了解决 LinkedIn 数据管道问题应运而生的。它的设计目的是提供一个高性能的消息系统，可以处理多种数据类型，并能够实时提供纯净且结构化的用户活动数据和系统度量指标。

> 数据为我们所做的每一件事提供了动力。
>
> ——Jeff Weiner，LinkedIn CEO

1.5.1 LinkedIn的问题

本章开头提到过，LinkedIn 有一个数据收集系统和应用程序指标，它使用自定义的收集器和一些开源工具来保存和展示内部数据。除了跟踪 CPU 使用率和应用性能这些一般性指标外，LinkedIn 还有一个比较复杂的用户请求跟踪功能。它使用了监控系统，可以跟踪单个用户的请求是如何在内部应用间传播的。不过监控系统存在很多不足。它使用的是轮询拉取度量指标的方式，指标之间的时间间隔较长，而且没有自助服务能力。它使用起来不太方便，很多简单的任务需要人工介入才能完成，而且一致性较差，同一个度量指标的名字在不同系统里的叫法不一样。

与此同时，我们还创建了另一个用于收集用户活动信息的系统。这是一个 HTTP 服务，前端的服务器会定期连接进来，在上面发布一些消息（XML 格式）。这些消息文件被转移到线下进行解析和校对。同样，这个系统也存在很多不足。XML 文件的格式无法保持一致，而且解析 XML 文件非常耗费计算资源。要想更改所创建的活动类型，需要在前端应用和离线处理程序之间做大量的协调工作。即使是这样，在更改数据结构时，仍然经常出现系统崩溃现象。而且批处理时间以小时计算，无法用它完成实时的任务。

监控和用户活动跟踪无法使用同一个后端服务。监控服务太过笨重，数据格式不适用于活动跟踪，而且无法在活动跟踪中使用轮询拉取模型。另一方面，把跟踪服务用在度量指标上也过于脆弱，批处理模型不适用于实时的监控和告警。不过，好在数据间存在很多共性，信息（比如特定类型的用户活动对应用程序性能的影响）之间的关联度还是很高的。特定类型用户活动数量的下降说明相关应用程序存在问题，不过批处理的长时间延迟意味着无法对这类问题作出及时的反馈。

最开始，我们调研了一些现成的开源解决方案，希望能够找到一个系统，可以实时访问数据，并通过横向扩展来处理大量的消息。我们使用 ActiveMQ 创建了一个原型系统，但它当时还无法满足横向扩展的需求。LinkedIn 不得不使用这种脆弱的解决方案，虽然 ActiveMQ 有很多缺陷会导致 broker 暂停服务。客户端的连接因此被阻塞，处理用户请求的能力也受到影响。于是我们最后决定构建自己的基础设施。

1.5.2　Kafka的诞生

LinkedIn 的开发团队由 Jay Kreps 领导。Jay Kreps 是 LinkedIn 的首席工程师，之前负责分布式键值存储系统 Voldemort 的开发。初建团队成员还包括 Neha Narkhede，不久之后， Jun Rao 也加入了进来。他们一起着手创建一个消息系统，可以同时满足上述的两种需求，并且可以在未来进行横向扩展。他们的主要目标如下：

- 使用推送和拉取模型解耦生产者和消费者；
- 为消息传递系统中的消息提供数据持久化，以便支持多个消费者；
- 通过系统优化实现高吞吐量；
- 系统可以随着数据流的增长进行横向扩展。

最后我们看到的这个发布与订阅消息系统具有典型的消息系统接口，但从存储层来看，它更像是一个日志聚合系统。Kafka 使用 Avro 作为消息序列化框架，每天高效地处理数十亿级别的度量指标和用户活动跟踪信息。LinkedIn 已经拥有超过万亿级别的消息使用量（截止到 2015 年 8 月），而且每天仍然需要处理超过千万亿字节的数据。

1.5.3　走向开源

2010 年底，Kafka 作为开源项目在 GitHub 上发布。2011 年 7 月，因为倍受开源社区的关注，它成为 Apache 软件基金会的孵化器项目。2012 年 10 月，Kafka 从孵化器项目毕业。从那时起，来自 LinkedIn 内部的开发团队一直为 Kafka 提供大力支持，而且吸引了大批来自 LinkedIn 外部的贡献者和参与者。现在，Kafka 被很多组织用在一些大型的数据管道上。2014 年秋天，Jay Kreps、Neha Narkhede 和 Jun Rao 离开 LinkedIn，创办了 Confluent。 Confluent 是一个致力于为企业开发提供支持、为 Kafka 提供培训的公司。这两家公司连同来自开源社区持续增长的贡献力量，一直在开发和维护 Kafka，让 Kafka 成为大数据管道的不二之选。

1.5.4　命名

关于 Kafka 的历史，人们经常会问到的一个问题就是，Kafka 这个名字是怎么想出来的，以及这个名字和这个项目之间有着怎样的联系。对于这个问题，Jay Kreps 解释如下：

> 我想既然 Kafka 是为了写数据而产生的，那么用作家的名字来命名会显得更有意义。我在大学时期上过很多文学课程，很喜欢 Franz Kafka。况且，对于开源项目来说，这个名字听起来很酷。因此，名字和应用本身基本没有太多联系。

1.6　开始Kafka之旅

现在我们对 Kafka 已经有了一个大体的了解，还知道了一些常见的术语，接下来可以开始使用 Kafka 来创建数据管道了。在下一章，我们将探究如何安装和配置 Kafka，还会讨论如何选择合适的硬件来运行 Kafka，以及把 Kafka 应用到生产环境需要注意的事项。

第 2 章　安装 Kafka

这一章将介绍如何安装和运行 Kafka，包括如何设置 Zookeeper（Kafka 使用 Zookeeper 保存 Broker 的元数据），还会介绍 Kafka 的基本配置，以及如何为 Kafka 选择合适的硬件，最后介绍如何在一个集群中安装多个 Kafka broker，以及把 Kafka 应用到生产环境需要注意的事项。

2.1　要事先行

在使用 Kafka 之前需要先做一些事情，接下来介绍怎样做。

2.1.1　选择操作系统

Kafka 是使用 Java 开发的应用程序，所以它可以运行在 Windows、MacOS 和 Linux 等多种操作系统上。本章将着重介绍如何在 Linux 上安装和使用 Kafka，因为把 Kafka 安装在 Linux 系统上是最为常见

的。即使只是把 Kafka 作为一般性用途，仍然推荐使用 Linux 系统。关于如何在 Windows 和 MacOS 上安装 Kafka，请参考附录 A。

2.1.2　安装Java

在安装 Zookeeper 和 Kafka 之前，需要先安装 Java 环境。这里推荐安装 Java 8，可以使用系统自带的安装包，也可以直接从 java.com 网站下载。虽然运行 Zookeeper 和 Kafka 只需要 Java 运行时版本，但也可以安装完整的 JDK，以备不时之需。假设 JDK 8 update 51 已经安装在 /usr/java/jdk1.8.0_51 目录下，其他软件的安装都是基于这个前提进行的。

2.1.3　安装Zookeeper

Kafka 使用 Zookeeper 保存集群的元数据信息和消费者信息。Kafka 发行版自带了 Zookeeper，可以直接从脚本启动，不过安装一个完整版的 Zookeeper 也并不费劲。

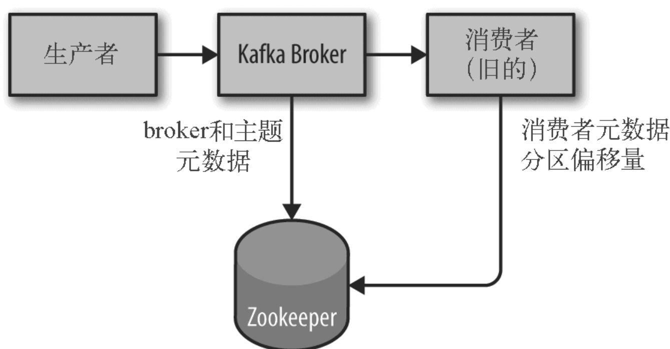


图 2-1：Kafka 和 Zookeeper

Zookeeper 的 3.4.6 稳定版已经在 Kafka 上做过全面测试，可以从 apache.org 下载该版本的 Zookeeper：http://bit.ly/2sDWSgJ。

01. 单机服务

    下面的例子演示了如何使用基本的配置安装 Zookeeper，安装目录为 /usr/local/zookeeper，数据目录为 /var/lib/zookeeper。

```
# tar -zxf zookeeper-3.4.6.tar.gz
```

```
# mv zookeeper-3.4.6 /usr/local/zookeeper
# mkdir -p /var/lib/zookeeper
# cat > /usr/local/zookeeper/conf/zoo.cfg << EOF
> tickTime=2000
> dataDir=/var/lib/zookeeper
> clientPort=2181
> EOF
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# /usr/local/zookeeper/bin/zkServer.sh start
JMX enabled by default
Using config: /usr/local/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
#
```

现在可以连到 Zookeeper 端口上，通过发送四字命令 srvr 来验证 Zookeeper 是否安装正确。

```
# telnet localhost 2181
Trying ::1...
Connected to localhost.
Escape character is '^]'.
srvr
Zookeeper version: 3.4.6-1569965, built on 02/20/2014 09:09 GMT
Latency min/avg/max: 0/0/0
Received: 1
Sent: 0
Connections: 1
Outstanding: 0
Zxid: 0x0
Mode: standalone
Node count: 4
Connection closed by foreign host.
#
```

02. **Zookeeper 群组（Ensemble）**

Zookeeper 集群被称为群组。Zookeeper 使用的是一致性协议，所以建议每个群组里应该包含奇数个节点（比如 3 个、5 个等），因为只有当群组里的大多数节点（也就是法定人数）处于可用状态，Zookeeper 才能处理外部的请求。也就是说，如果你有一个包含 3 个节点的群组，那么它允许一个节点失效。如果群组包含 5 个节点，那么它允许 2 个节点失效。

群组节点个数的选择

假设有一个包含 5 个节点的群组，如果要对群

组做一些包括更换节点在内的配置更改，需要依次重启每一个节点。如果你的群组无法容忍多个节点失效，那么在进行群组维护时就会存在风险。不过，也不建议一个群组包含超过 7 个节点，因为 Zookeeper 使用了一致性协议，节点过多会降低整个群组的性能。

群组需要有一些公共配置，上面列出了所有服务器的清单，并且每个服务器还要在数据目录中创建一个 myid 文件，用于指明自己的 ID。如果群组里服务器的机器名是 zoo1.example.com、zoo2.example.com、zoo3.example.com，那么配置文件可能是这样的：

```
tickTime=2000
dataDir=/var/lib/zookeeper
clientPort=2181
initLimit=20
syncLimit=5
server.1=zoo1.example.com:2888:3888
server.2=zoo2.example.com:2888:3888
server.3=zoo3.example.com:2888:3888
```

在这个配置中，`initLimit` 表示用于在从节点与主节点之间建立初始化连接的时间上限，`syncLimit` 表示允许从节点与主节点处于不同步状态的时间上限。这两个值都是 `tickTime` 的倍数，所以 `initLimit` 是 20*2000ms，也就是 40s。配置里还列出了群组中所有服务器的地址。服务器地址遵循 `server.X=hostname:peerPort:leaderPort` 格式，各个参数说明如下：

X

服务器的 ID，它必须是一个整数，不过不一定要从 0 开始，也不要求是连续的；

hostname

服务器的机器名或 IP 地址；

peerPort

用于节点间通信的 TCP 端口；

leaderPort

用于首领选举的 TCP 端口。

客户端只需要通过 clientPort 就能连接到群组，而群组节点间的通信则需要同时用到这 3 个端口（peerPort、leaderPort、clientPort）。

除了公共的配置文件外，每个服务器都必须在 data Dir 目录中创建一个叫作 myid 的文件，文件里要包含服务器 ID，这个 ID 要与配置文件里配置的 ID 保持一致。完成这些步骤后，就可以启动服务器，让它们彼此间进行通信了。

2.2　安装Kafka Broker

配置好 Java 和 Zookeeper 之后，接下来就可以安装 Kafka 了。可以从 http://kafka.apache.org/downloads.html 下载最新版本的 Kafka。截至本书写作时，Kafka 的版本是 0.9.0.1，对应的 Scala 版本是 2.11.0。

下面的例子将 Kafka 安装在 /usr/local/kafka 目录下，使用之前配置好的 Zookeeper，并把消息日志保存在 /tmp/kafka-logs 目录下。

```
# tar -zxf kafka_2.11-0.9.0.1.tgz
# mv kafka_2.11-0.9.0.1 /usr/local/kafka
# mkdir /tmp/kafka-logs
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

一旦 Kafka 创建完毕，就可以对这个集群做一些简单的操作来验证它是否安装正确，比如创建一个测试主题，发布一些消息，然后读取它们。

创建并验证主题：

```
# /usr/local/kafka/bin/kafka-topics.sh --create --zookeeper localhost:2181
--replication-factor 1 --partitions 1 --topic test
Created topic "test".
# /usr/local/kafka/bin/kafka-topics.sh --zookeeper localhost:2181
--describe --topic test
```

```
Topic:test    PartitionCount:1    ReplicationFactor:1    Configs:
    Topic: test    Partition: 0    Leader: 0    Replicas: 0    Isr: 0
#
```

往测试主题上发布消息：

```
# /usr/local/kafka/bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic test
Test Message 1
Test Message 2
^D
#
```

从测试主题上读取消息：

```
# /usr/local/kafka/bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic test --from-beginning
Test Message 1
Test Message 2
^C
Consumed 2 messages
#
```

2.3　broker配置

Kafka 发行包里自带的配置样本可以用来安装单机服务，但并不能满足大多数安装场景的要求。Kafka 有很多配置选项，涉及安装和调优的方方面面。不过大多数调优选项可以使用默认配置，除非你对调优有特别的要求。

2.3.1　常规配置

有一些配置选项，在单机安装时可以直接使用默认值，但在部署到其他环境时要格外小心。这些参数是单个服务器最基本的配置，它们中的大部分需要经过修改后才能用在集群里。

01. **broker.id**

    每个 broker 都需要有一个标识符，使用 broker.id 来表示。它的默认值是 0，也可以被设置成其他任意整数。这个值在整个 Kafka 集群里必须是唯一的。这个值可以任意选定，如果出于

维护的需要，可以在服务器节点间交换使用这些 ID。建议把它们设置成与机器名具有相关性的整数，这样在进行维护时，将 ID 号映射到机器名就没那么麻烦了。例如，如果机器名包含唯一性的数字（比如 host1.example.com、host2.example.com），那么用这些数字来设置 broker.id 就再好不过了。

02. **port**

如果使用配置样本来启动 Kafka，它会监听 9092 端口。修改 `port` 配置参数可以把它设置成其他任意可用的端口。要注意，如果使用 1024 以下的端口，需要使用 root 权限启动 Kafka，不过不建议这么做。

03. **zookeeper.connect**

用于保存 broker 元数据的 Zookeeper 地址是通过 `zookeeper.connect` 来指定的。`localhost:2181` 表示这个 Zookeeper 是运行在本地的 2181 端口上。该配置参数是用冒号分隔的一组 `hostname:port/path` 列表，每一部分的含义如下：

- `hostname` 是 Zookeeper 服务器的机器名或 IP 地址；
- `port` 是 Zookeeper 的客户端连接端口；
- /path 是可选的 Zookeeper 路径，作为 Kafka 集群的 chroot 环境。如果不指定，默认使用根路径。

如果指定的 chroot 路径不存在，broker 会在启动的时候创建它。

为什么使用 **chroot** 路径

在 Kafka 集群里使用 chroot 路径是一种最佳实践。Zookeeper 群组可以共享给其他应用程序，即使还有其他 Kafka 集群存在，也不会产生冲突。最好是在配置文件里指定一组 Zookeeper 服务器，用分号把它们隔开。一旦有一个 Zookeeper 服务器宕机，broker 可以连接到 Zookeeper 群组的另一个节点上。

07. **log.dirs**

Kafka 把所有消息都保存在磁盘上，存放这些日志片段的目录是通过 log.dirs 指定的。它是一组用逗号分隔的本地文件系统路径。如果指定了多个路径，那么 broker 会根据“最少使用”原则，把同一个分区的日志片段保存到同一个路径下。要注意，broker 会往拥有最少数目分区的路径新增分区，而不是往拥有最小磁盘空间的路径新增分区。

08. **num.recovery.threads.per.data.dir**

对于如下 3 种情况，Kafka 会使用可配置的线程池来处理日志片段：

- 服务器正常启动，用于打开每个分区的日志片段；
- 服务器崩溃后重启，用于检查和截短每个分区的日志片段；
- 服务器正常关闭，用于关闭日志片段。

默认情况下，每个日志目录只使用一个线程。因为这些线程只是在服务器启动和关闭时会用到，所以完全可以设置大量的线程来达到并行操作的目的。特别是对于包含大量分区的服务器来说，一旦发生崩溃，在进行恢复时使用并行操作可能会省下数小时的时间。设置此参数时需要注意，所配置的数字对应的是 log.dirs 指定的单个日志目录。也就是说，如果 `num.recovery.threads.per.data.dir` 被设为 8，并且 `log.dir` 指定了 3 个路径，那么总共需要 24 个线程。

12. **auto.create.topics.enable**

默认情况下，Kafka 会在如下几种情形下自动创建主题：

- 当一个生产者开始往主题写入消息时；
- 当一个消费者开始从主题读取消息时；
- 当任意一个客户端向主题发送元数据请求时。

很多时候，这些行为都是非预期的。而且，根据 Kafka 协议，如果一个主题不先被创建，根本无法知道它是否已经存在。如果显式地创建主题，不管是手动创建还是通过其他配置系统来创建，都可以把 `auto.create.topics.enable` 设为 `false`。

2.3.2　主题的默认配置

Kafka 为新创建的主题提供了很多默认配置参数。可以通过管理工具（将在第 9 章介绍）为每个主题单独配置一部分参数，比如分区个数和数据保留策略。服务器提供的默认配置可以作为基准，它们适用于大部分主题。

> **使用主题配置覆盖（override）**
>
> 之前的 Kafka 版本允许主题覆盖服务器的默认配置，包括 `log.retention.hours.per.topic`、`log.retention.bytes.per.topic` 和 `log.seg ment.bytes.per.topic` 这几个参数。新版本不再支持这些参数，而且如果要对参数进行覆盖，需要使用管理工具。

01. **num.partitions**

 `num.partitions` 参数指定了新创建的主题将包含多少个分区。如果启用了主题自动创建功能（该功能默认是启用的），主题分区的个数就是该参数指定的值。该参数的默认值是 1。要注意，我们可以增加主题分区的个数，但不能减少分区的个数。所以，如果要让一个主题的分区个数少于 `num.partitions` 指定的值，需要手动创建该主题（将在第 9 章讨论）。

 第 1 章里已经提到，Kafka 集群通过分区对主题进行横向扩展，所以当有新的 broker 加入集群时，可以通过分区个数来实现集群的负载均衡。当然，这并不是说，在存在多个主题的情况下（它们分布在多个 broker 上），为了能让分区分布到所有 broker 上，主题分区的个数必须要大于 broker 的个数。不过，拥有大量消息的主题如果要进行负载分散，就需要大量的分区。

 > 如何选定分区数量
 >
 > 为主题选定分区数量并不是一件可有可无的事情，在进行数量选择时，需要考虑如下几个因素。
 >
 > - 主题需要达到多大的吞吐量？例如，是希望每秒钟写入 100KB 还是 1GB ？
 > - 从单个分区读取数据的最大吞吐量是多

少？每个分区一般都会有一个消费者，如果你知道消费者将数据写入数据库的速度不会超过每秒 50MB，那么你也该知道，从一个分区读取数据的吞吐量不需要超过每秒 50MB。

- 可以通过类似的方法估算生产者向单个分区写入数据的吞吐量，不过生产者的速度一般比消费者快得多，所以最好为生产者多估算一些吞吐量。
- 每个 broker 包含的分区个数、可用的磁盘空间和网络带宽。
- 如果消息是按照不同的键来写入分区的，那么为已有的主题新增分区就会很困难。
- 单个 broker 对分区个数是有限制的，因为分区越多，占用的内存越多，完成首领选举需要的时间也越长。

很显然，综合考虑以上几个因素，你需要很多分区，但不能太多。如果你估算出主题的吞吐量和消费者吞吐量，可以用主题吞吐量除以消费者吞吐量算出分区的个数。也就是说，如果每秒钟要从主题上写入和读取 1GB 的数据，并且每个消费者每秒钟可以处理 50MB 的数据，那么至少需要 20 个分区。这样就可以让 20 个消费者同时读取这些分区，从而达到每秒钟 1GB 的吞吐量。

如果不知道这些信息，那么根据经验，把分区的大小限制在 25GB 以内可以得到比较理想的效果。

08. **log.retention.ms**

Kafka 通常根据时间来决定数据可以被保留多久。默认使用 `log.retention.hours` 参数来配置时间，默认值为 168 小时，也就是一周。除此以外，还有其他两个参数 `log.retention.minutes` 和 `log.retention.ms`。这 3 个参数的作用是一样的，都是决定消息多久以后会被删除，不过还是推荐使用 `log.retention.ms`。如果指定了不止一个参数，Kafka 会优先使用具有最小值的那个参数。

根据时间保留数据和最后修改时间

根据时间保留数据是通过检查磁盘上日志片段文件的最后修改时间来实现的。一般来说，最后修改时间指的就是日志片段的关闭时间，也就是文件里最后一个消息的时间戳。不过，如果使用管理工具在服务器间移动分区，最后修改时间就不准确了。时间误差可能导致这些分区过多地保留数据。在第 9 章讨论分区移动时会提到更多这方面的内容。

09. **log.retention.bytes**

另一种方式是通过保留的消息字节数来判断消息是否过期。它的值通过参数 `log.retention.bytes` 来指定，作用在每一个分区上。也就是说，如果有一个包含 8 个分区的主题，并且 `log.retention.bytes` 被设为 1GB，那么这个主题最多可以保留 8GB 的数据。所以，当主题的分区个数增加时，整个主题可以保留的数据也随之增加。

根据字节大小和时间保留数据

如果同时指定了 `log.retention.bytes` 和 `log.retention.ms`（或者另一个时间参数），只要任意一个条件得到满足，消息就会被删除。例如，假设 `log.retention.ms` 设置为 86 400 000（也就是 1 天），`log.retention.bytes` 设置为 1 000 000 000（也就是 1GB），如果消息字节总数在不到一天的时间就超过了 1GB，那么多出来的部分就会被删除。相反，如果消息字节总数小于 1GB，那么一天之后这些消息也会被删除，尽管分区的数据总量小于 1GB。

10. **log.segment.bytes**

以上的设置都作用在日志片段上，而不是作用在单个消息上。当消息到达 broker 时，它们被追加到分区的当前日志片段上。当日志片段大小达到 `log.segment.bytes` 指定的上限（默认是 1GB）时，当前日志片段就会被关闭，一个新的日志片段被打开。如果一个日志片段被关闭，就开始等待过期。这个参数的值越小，就会越频繁地关闭和分配新文件，从而降低磁盘写入的整体效率。

如果主题的消息量不大，那么如何调整这个参数的大小就变得尤为重要。例如，如果一个主题每天只接收 100MB 的消息，而 `log.segment.bytes` 使用默认设置，那么需要 10 天时间才能填满一个日志片段。因为在日志片段被关闭之前消息是不会过期的，所以如果 `log.retention.ms` 被设为 604 800 000（也就是 1 周），那么日志片段最多需要 17 天才会过期。

这是因为关闭日志片段需要 10 天的时间，而根据配置的过期时间，还需要再保留 7 天时间（要等到日志片段里的最后一个消息过期才能被删除）。

> 使用时间戳获取偏移量
>
> 日志片段的大小会影响使用时间戳获取偏移量。在使用时间戳获取日志偏移量时，Kafka 会检查分区里最后修改时间大于指定时间戳的日志片段（已经被关闭的），该日志片段的前一个文件的最后修改时间小于指定时间戳。然后，Kafka 返回该日志片段（也就是文件名）开头的偏移量。对于使用时间戳获取偏移量的操作来说，日志片段越小，结果越准确。

11. **log.segment.ms**

另一个可以控制日志片段关闭时间的参数是 `log.segment.ms`，它指定了多长时间之后日志片段会被关闭。就像 `log.retention.bytes` 和 `log.retention.ms` 这两个参数一样，`log.segment.bytes` 和 `log.retention.ms` 这两个参数之间也不存在互斥问题。日志片段会在大小或时间达到上限时被关闭，就看哪个条件先得到满足。默认情况下，`log.segment.ms` 没有设定值，所以只根据大小来关闭日志片段。

> 基于时间的日志片段对磁盘性能的影响
>
> 在使用基于时间的日志片段时，要着重考虑并行关闭多个日志片段对磁盘性能的影响。如果多个分区的日志片段永远不能达到大小的上限，就会发生这种情况，因为 broker 在启动之后就开始计算日志片段的过期时间，对于那些数据量小的分区来说，日志片段的关闭操作

总是同时发生。

12. **message.max.bytes**

broker 通过设置 `message.max.bytes` 参数来限制单个消息的大小，默认值是 1 000 000，也就是 1MB。如果生产者尝试发送的消息超过这个大小，不仅消息不会被接收，还会收到 broker 返回的错误信息。跟其他与字节相关的配置参数一样，该参数指的是压缩后的消息大小，也就是说，只要压缩后的消息小于 `message.max.bytes` 指定的值，消息的实际大小可以远大于这个值。

这个值对性能有显著的影响。值越大，那么负责处理网络连接和请求的线程就需要花越多的时间来处理这些请求。它还会增加磁盘写入块的大小，从而影响 IO 吞吐量。

> 在服务端和客户端之间协调消息大小的配置
>
> 消费者客户端设置的 `fetch.message.max.bytes` 必须与服务器端设置的消息大小进行协调。如果这个值比 `message.max.bytes` 小，那么消费者就无法读取比较大的消息，导致出现消费者被阻塞的情况。在为集群里的 broker 配置 `replica.fetch.max.bytes` 参数时，也遵循同样的原则。

2.4 硬件的选择

为 Kafka 选择合适的硬件更像是一门艺术。Kafka 本身对硬件没有特别的要求，它可以运行在任何系统上。不过，如果比较关注性能，那么就需要考虑几个会影响整体性能的因素：磁盘吞吐量和容量、内存、网络和 CPU。在确定了性能关注点之后，就可以在预算范围内选择最优化的硬件配置。

2.4.1 磁盘吞吐量

生产者客户端的性能直接受到服务器端磁盘吞吐量的影响。生产者生成的消息必须被提交到服务器保存，大多数客户端在发送消息之后会

一直等待，直到至少有一个服务器确认消息已经成功提交为止。也就是说，磁盘写入速度越快，生成消息的延迟就越低。

在考虑硬盘类型对磁盘吞吐量的影响时，是选择传统的机械硬盘（HDD）还是固态硬盘（SSD），我们可以很容易地作出决定。固态硬盘的查找和访问速度都很快，提供了最好的性能。机械硬盘更便宜，单块硬盘容量也更大。在同一个服务器上使用多个机械硬盘，可以设置多个数据目录，或者把它们设置成磁盘阵列，这样可以提升机械硬盘的性能。其他方面的因素，比如磁盘特定的技术（串行连接存储技术或 SATA），或者磁盘控制器的质量，都会影响吞吐量。

2.4.2　磁盘容量

磁盘容量是另一个值得讨论的话题。需要多大的磁盘容量取决于需要保留的消息数量。如果服务器每天会收到 1TB 消息，并且保留 7 天，那么就需要 7TB 的存储空间，而且还要为其他文件提供至少 10% 的额外空间。除此之外，还需要提供缓冲区，用于应付消息流量的增长和波动。

在决定扩展 Kafka 集群规模时，存储容量是一个需要考虑的因素。通过让主题拥有多个分区，集群的总流量可以被均衡到整个集群，而且如果单个 broker 无法支撑全部容量，可以让其他 broker 提供可用的容量。存储容量的选择同时受到集群复制策略的影响（将在第 6 章讨论更多的细节）。

2.4.3　内存

除了磁盘性能外，服务器端可用的内存容量是影响客户端性能的主要因素。磁盘性能影响生产者，而内存影响消费者。消费者一般从分区尾部读取消息，如果有生产者存在，就紧跟在生产者后面。在这种情况下，消费者读取的消息会直接存放在系统的页面缓存里，这比从磁盘上重新读取要快得多。

运行 Kafka 的 JVM 不需要太大的内存，剩余的系统内存可以用作页面缓存，或者用来缓存正在使用中的日志片段。这也就是为什么不建议把 Kafka 同其他重要的应用程序部署在一起的原因，它们需要共享页面缓存，最终会降低 Kafka 消费者的性能。

2.4.4　网络

网络吞吐量决定了 Kafka 能够处理的最大数据流量。它和磁盘存储是

制约 Kafka 扩展规模的主要因素。Kafka 支持多个消费者，造成流入和流出的网络流量不平衡，从而让情况变得更加复杂。对于给定的主题，一个生产者可能每秒钟写入 1MB 数据，但可能同时有多个消费者瓜分网络流量。其他的操作，如集群复制（在第 6 章介绍）和镜像（在第 8 章介绍）也会占用网络流量。如果网络接口出现饱和，那么集群的复制出现延时就在所难免，从而让集群不堪一击。

2.4.5 CPU

与磁盘和内存相比，Kafka 对计算处理能力的要求相对较低，不过它在一定程度上还是会影响整体的性能。客户端为了优化网络和磁盘空间，会对消息进行压缩。服务器需要对消息进行批量解压，设置偏移量，然后重新进行批量压缩，再保存到磁盘上。这就是 Kafka 对计算处理能力有所要求的地方。不过不管怎样，这都不应该成为选择硬件的主要考虑因素。

2.5 云端的Kafka

Kafka 一般被安装在云端，比如亚马逊网络服务（Amazon Web Services，AWS）。AWS 提供了很多不同配置的实例，我们要根据 Kafka 的性能优先级来选择合适的实例。可以先从要保留数据的大小开始考虑，然后考虑生产者方面的性能。如果要求低延迟，那么就需要专门为 I/O 优化过的使用固态硬盘的实例，否则，使用配备了临时存储的实例就可以了。选好存储类型之后，再选择 CPU 和内存就容易得多。

实际上，如果使用 AWS，一般会选择 m4 实例或 r3 实例。m4 实例允许较长时间地保留数据，不过磁盘吞吐量会小一些，因为它使用的是弹性块存储。r3 实例使用固态硬盘，具有较高的吞吐量，但保留的数据量会有所限制。如果想两者兼顾，那么需要升级成 i2 实例或 d2 实例，不过它们的成本要高得多。

2.6 Kafka集群

单个 Kafka 服务器足以满足本地开发或 POC 要求，不过集群也有它的强大之处。使用集群最大的好处是可以跨服务器进行负载均衡，再则就是可以使用复制功能来避免因单点故障造成的数据丢失。在维护 Kafka 或底层系统时，使用集群可以确保为客户端提供高可用性。本节只是介绍如何配置 Kafka 集群，第 6 章将介绍更多关于数据复制的

内容。

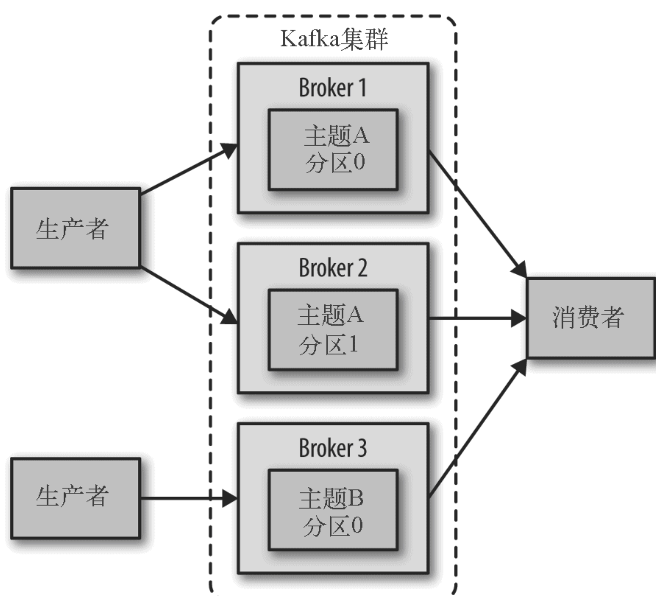


图 **2-2**：一个简单的 **Kafka** 集群

2.6.1　需要多少个broker

一个 Kafka 集群需要多少个 broker 取决于以下几个因素。首先，需要多少磁盘空间来保留数据，以及单个 broker 有多少空间可用。如果整个集群需要保留 10TB 的数据，每个 broker 可以存储 2TB，那么至少需要 5 个 broker。如果启用了数据复制，那么至少还需要一倍的空间，不过这要取决于配置的复制系数是多少（将在第 6 章介绍）。也就是说，如果启用了数据复制，那么这个集群至少需要 10 个 broker。

第二个要考虑的因素是集群处理请求的能力。这通常与网络接口处理客户端流量的能力有关，特别是当有多个消费者存在或者在数据保留期间流量发生波动（比如高峰时段的流量爆发）时。如果单个 broker 的网络接口在高峰时段可以达到 80% 的使用量，并且有两个消费者，那么消费者就无法保持峰值，除非有两个 broker。如果集群启用了复制功能，则要把这个额外的消费者考虑在内。因磁盘吞吐量低和系统内存不足造成的性能问题，也可以通过扩展多个 broker 来解决。

2.6.2 broker配置

要把一个 broker 加入到集群里，只需要修改两个配置参数。首先，所有 broker 都必须配置相同的 `zookeeper.connect`，该参数指定了用于保存元数据的 Zookeeper 群组和路径。其次，每个 broker 都必须为 `broker.id` 参数设置唯一的值。如果两个 broker 使用相同的 `broker.id`，那么第二个 broker 就无法启动。在运行集群时，还可以配置其他一些参数，特别是那些用于控制数据复制的参数，这些将在后续的章节介绍。

2.6.3 操作系统调优

大部分 Linux 发行版默认的内核调优参数配置已经能够满足大多数应用程序的运行需求，不过还是可以通过调整一些参数来进一步提升 Kafka 的性能。这些参数主要与虚拟内存、网络子系统和用来存储日志片段的磁盘挂载点有关。这些参数一般配置在 /etc/sysctl.conf 文件里，不过在对内核参数进行调整时，最好参考操作系统的文档。

01. 虚拟内存

    一般来说，Linux 的虚拟内存会根据系统的工作负荷进行自动调整。我们可以对交换分区的处理方式和内存脏页进行调整，从而让 Kafka 更好地处理工作负载。

    对于大多数依赖吞吐量的应用程序来说，要尽量避免内存交换。内存页和磁盘之间的交换对 Kafka 各方面的性能都有重大影响。Kafka 大量地使用系统页面缓存，如果虚拟内存被交换到磁盘，说明已经没有多余内存可以分配给页面缓存了。

    一种避免内存交换的方法是不设置任何交换分区。内存交换不是必需的，不过它确实能够在系统发生灾难性错误时提供一些帮助。进行内存交换可以防止操作系统由于内存不足而突然终止进程。基于上述原因，建议把 `vm.swappiness` 参数的值设置得小一点，比如 1。该参数指明了虚拟机的子系统将如何使用交换分区，而不是只把内存页从页面缓存里移除。要优先考虑减小页面缓存，而不是进行内存交换。

> **为什么不把 `vm.swappiness` 设为零**
>
> 先前，人们建议尽量把 `vm.swapiness` 设为 0，它意味着“除非发生内存溢出，否则不要进

行内存交换”。直到 Linux 内核 3.5-rc1 版本发布，这个值的意义才发生了变化。这个变化被移植到其他的发行版上，包括 Red Hat 企业版内核 2.6.32-303。在发生变化之后，0 意味着“在任何情况下都不要发生交换”。所以现在建议把这个值设为 1。

脏页会被冲刷到磁盘上，调整内核对脏页的处理方式可以让我们从中获益。Kafka 依赖 I/O 性能为生产者提供快速的响应。这就是为什么日志片段一般要保存在快速磁盘上，不管是单个快速磁盘（如 SSD）还是具有 NVRAM 缓存的磁盘子系统（如 RAID）。这样一来，在后台刷新进程将脏页写入磁盘之前，可以减少脏页的数量，这个可以通过将 `vm.dirty_background_ratio` 设为小于 10 的值来实现。该值指的是系统内存的百分比，大部分情况下设为 5 就可以了。它不应该被设为 0，因为那样会促使内核频繁地刷新页面，从而降低内核为底层设备的磁盘写入提供缓冲的能力。

通过设置 `vm.dirty_ratio` 参数可以增加被内核进程刷新到磁盘之前的脏页数量，可以将它设为大于 20 的值（这也是系统内存的百分比）。这个值可设置的范围很广，60~80 是个比较合理的区间。不过调整这个参数会带来一些风险，包括未刷新磁盘操作的数量和同步刷新引起的长时间 I/O 等待。如果该参数设置了较高的值，建议启用 Kafka 的复制功能，避免因系统崩溃造成数据丢失。

为了给这些参数设置合适的值，最好是在 Kafka 集群运行期间检查脏页的数量，不管是在生存环境还是模拟环境。可以在 /proc/vmstat 文件里查看当前脏页数量。

```
# cat /proc/vmstat | egrep "dirty|writeback"
nr_dirty 3875
nr_writeback 29
nr_writeback_temp 0
#
```

02. 磁盘

除了选择合适的磁盘硬件设备和使用 RAID 外，文件系统是影响性能的另一个重要因素。有很多种文件系统可供选择，不过对于本地文件系统来说，EXT4（第四代可扩展文件系统）和

XFS 最为常见。近来，XFS 成为很多 Linux 发行版默认的文件系统，因为它只需要做少量调优就可以承担大部分的工作负荷，比 EXT4 具有更好的表现。EXT4 也可以做得很好，但需要做更多的调优，存在较大的风险。其中就包括设置更长的提交间隔（默认是 5），以便降低刷新的频率。EXT4 还引入了块分配延迟，一旦系统崩溃，更容易造成数据丢失和文件系统毁坏。XFS 也使用了分配延迟算法，不过比 EXT4 的要安全些。XFS 为 Kafka 提供了更好的性能，除了由文件系统提供的自动调优之外，无需额外的调优。批量磁盘写入具有更高的效率，可以提升整体的 I/O 吞吐量。

不管使用哪一种文件系统来存储日志片段，最好要对挂载点的 `noatime` 参数进行合理的设置。文件元数据包含 3 个时间戳：创建时间（ctime）、最后修改时间（mtime）以及最后访问时间（atime）。默认情况下，每次文件被读取后都会更新 atime，这会导致大量的磁盘写操作，而且 atime 属性的用处不大，除非某些应用程序想要知道某个文件在最近一次修改后有没有被访问过（这种情况可以使用 `realtime`）。Kafka 用不到该属性，所以完全可以把它禁用掉。为挂载点设置 `noatime` 参数可以防止更新 atime，但不会影响 ctime 和 mtime。

03. 网络

默认情况下，系统内核没有针对快速的大流量网络传输进行优化，所以对于应用程序来说，一般需要对 Linux 系统的网络栈进行调优，以实现对大流量的支持。实际上，调整 Kafka 的网络配置与调整其他大部分 Web 服务器和网络应用程序的网络配置是一样的。首先可以对分配给 socket 读写缓冲区的内存大小作出调整，这样可以显著提升网络的传输性能。socket 读写缓冲区对应的参数分别是 `net.core.wmem_default` 和 `net.core.rmem_default`，合理的值是 131 072（也就是 128KB）。读写缓冲区最大值对应的参数分别是 `net.core.wmem_max` 和 `net.core.rmem_max`，合理的值是 2 097 152（也就是 2MB）。要注意，最大值并不意味着每个 socket 一定要有这么大的缓冲空间，只是说在必要的情况下才会达到这个值。

除了设置 socket 外，还需要设置 TCP socket 的读写缓冲区，它们的参数分别是 `net.ipv4.tcp_wmem` 和 `net.ipv4.tcp_rmem`。这些参数的值由 3 个整数组成，它们

使用空格分隔，分别表示最小值、默认值和最大值。最大值不能大于 net.core.wmem_max 和 net.core.rmem_max 指定的大小。例如，“4096 65536 2048000”表示最小值是 4KB、默认值是 64KB、最大值是 2MB。根据 Kafka 服务器接收流量的实际情况，可能需要设置更高的最大值，为网络连接提供更大的缓冲空间。

还有其他一些有用的网络参数。例如，把 net.ipv4.tcp_window_scaling 设为 1，启用 TCP 时间窗扩展，可以提升客户端传输数据的效率，传输的数据可以在服务器端进行缓冲。把 net.ipv4.tcp_max_syn_backlog 设为比默认值 1024 更大的值，可以接受更多的并发连接。把 net.core.netdev_max_backlog 设为比默认值 1000 更大的值，有助于应对网络流量的爆发，特别是在使用千兆网络的情况下，允许更多的数据包排队等待内核处理。

2.7 生产环境的注意事项

当你准备把 Kafka 从测试环境部署到生产环境时，需要注意一些事项，以便创建更可靠的消息服务。

2.7.1 垃圾回收器选项

为应用程序调整 Java 垃圾回收参数就像是一门艺术，我们需要知道应用程序是如何使用内存的，还需要大量的观察和试错。幸运的是，Java 7 为我们带来了 G1 垃圾回收器，让这种状况有所改观。在应用程序的整个生命周期，G1 会自动根据工作负载情况进行自我调节，而且它的停顿时间是恒定的。它可以轻松地处理大块的堆内存，把堆内存分为若干小块的区域，每次停顿时并不会对整个堆空间进行回收。

正常情况下，G1 只需要很少的配置就能完成这些工作。以下是 G1 的两个调整参数。

MaxGCPauseMillis：

该参数指定每次垃圾回收默认的停顿时间。该值不是固定的，G1 可以根据需要使用更长的时间。它的默认值是 200ms。也就是说，G1 会决定垃圾回收的频率以及每一轮需要回收多少个区域，这样算下来，每一轮垃圾回收大概需要 200ms 的时间。

InitiatingHeapOccupancyPercent：

该参数指定了在 G1 启动新一轮垃圾回收之前可以使用的堆内存百分比，默认值是 45。也就是说，在堆内存的使用率达到 45% 之前，G1 不会启动垃圾回收。这个百分比包括新生代和老年代的内存。

Kafka 对堆内存的使用率非常高，容易产生垃圾对象，所以可以把这些值设得小一些。如果一台服务器有 64GB 内存，并且使用 5GB 堆内存来运行 Kafka，那么可以参考以下的配置：MaxGCPauseMillis 可以设为 20ms；InitiatingHeapOccupancyPercent 可以设为 35，这样可以让垃圾回收比默认的要早一些启动。

Kafka 的启动脚本并没有启用 G1 回收器，而是使用了 Parallel New 和 CMS（ Concurrent Mark-Sweep，并发标记和清除）垃圾回收器。不过它可以通过环境变量来修改。本章前面的内容使用 start 命令来修改它：

```
# export JAVA_HOME=/usr/java/jdk1.8.0_51
# export KAFKA_JVM_PERFORMANCE_OPTS="-server -XX:+UseG1GC
-XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35
-XX:+DisableExplicitGC -Djava.awt.headless=true"
# /usr/local/kafka/bin/kafka-server-start.sh -daemon
/usr/local/kafka/config/server.properties
#
```

2.7.2　数据中心布局

在开发阶段，人们并不会太关心 Kafka 服务器在数据中心所处的物理位置，因为即使集群在短时间内出现局部或完全不可用，也不会造成太大影响。但是，在生产环境，服务不可用意味着金钱的损失，具体表现为无法为用户提供服务或者不知道用户正在做什么。这个时候，使用 Kafka 集群的复制功能就变得尤为重要（请参考第 6 章），而服务器在数据中心所处的物理位置也变得重要起来。如果在部署 Kafka 之前没有考虑好这个问题，那么在后续的维护过程中，移动服务器需要耗费更高的成本。

在为 broker 增加新的分区时，broker 并无法获知机架的信息。也就是说，两个 broker 有可能是在同一个机架上，或者在同一个可用区域里（如果运行在像 AWS 这样的的云服务上），所以，在为分区添

加副本的时候，这些副本很可能被分配给同一个机架上的 broker，它们使用相同的电源和网络连接。如果该机架出了问题，这些分区就会离线，客户端就无法访问到它们。更糟糕的是，如果发生不完整的主节点选举，那么在恢复时就有可能丢失数据（第 6 章将介绍更多细节）。

所以，最好把集群的 broker 安装在不同的机架上，至少不要让它们共享可能出现单点故障的基础设施，比如电源和网络。也就是说，部署服务器需要至少两个电源连接（两个不同的回路）和两个网络交换器（保证可以进行无缝的故障切换）。除了这些以外，最好还要把 broker 安放在不同的机架上。因为随着时间的推移，机架也需要进行维护，而这会导致机器离线（比如移动机器或者重新连接电源）。

2.7.3　共享Zookeeper

Kafka 使用 Zookeeper 来保存 broker、主题和分区的元数据信息。对于一个包含多个节点的 Zookeeper 群组来说，Kafka 集群的这些流量并不算多，那些写操作只是用于构造消费者群组或集群本身。实际上，在很多部署环境里，会让多个 Kafka 集群共享一个 Zookeeper 群组（每个集群使用一个 chroot 路径）。

> **Kafka 消费者和 Zookeeper**
>
> 在 Kafka 0.9.0.0 版本之前，除了 broker 之外，消费者也会使用 Zookeeper 来保存一些信息，比如消费者群组的信息、主题信息、消费分区的偏移量（在消费者群组里发生失效转移时会用到）。到了 0.9.0.0 版本，Kafka 引入了一个新的消费者接口，允许 broker 直接维护这些信息。这个新的消费者接口将在第 4 章介绍。

不过，消费者和 Zookeeper 之间还是有一个值得注意的地方，消费者可以选择将偏移量提交到 Zookeeper 或 Kafka，还可以选择提交偏移量的时间间隔。如果消费者将偏移量提交到 Zookeeper，那么在每个提交时间点上，消费者将会为每一个消费的分区往 Zookeeper 写入一次偏移量。合理的提交间隔是 1 分钟，因为这刚好是消费者群组的某个消费者发生失效时能够读取到重复消息的时间。值得注意的是，这些提交对于 Zookeeper 来说流量不算小，特别是当集群里有多个消费者的时候。如果 Zookeeper 群组无法处理太大的流量，就有必要使用长一点的提交时间间隔。不过不管怎样，还是建议使用最新版本的 Kafka，让消费者把偏移量提交到 Kafka 服务器上，消除对

Zookeeper 的依赖。

虽然多个 Kafka 集群可以共享一个 Zookeeper 群组，但如果有可能的话，不建议把 Zookeeper 共享给其他应用程序。Kafka 对 Zookeeper 的延迟和超时比较敏感，与 Zookeeper 群组之间的一个通信异常就可能导致 Kafka 服务器出现无法预测的行为。这样很容易让多个 broker 同时离线，如果它们与 Zookeeper 之间断开连接，也会导致分区离线。这也会给集群控制器带来压力，在服务器离线一段时间之后，当控制器尝试关闭一个服务器时，会表现出一些细小的错误。其他的应用程序因重度使用或进行不恰当的操作给 Zookeeper 群组带来压力，所以最好让它们使用自己的 Zookeeper 群组。

2.8　总结

在这一章，我们学习了如何运行 Kafka，同时也讨论了如何为 Kafka 选择合适的硬件，以及在生产环境中使用 Kafka 需要注意的事项。有了 Kafka 集群之后，接下来要介绍基本的客户端应用程序。后面两章将介绍如何创建客户端，并用它们向 Kafka 生产消息（第 3 章）以及从 Kafka 读取这些消息（第 4 章）。

第 3 章　Kafka 生产者——向 Kafka 写入数据

不管是把 Kafka 作为消息队列、消息总线还是数据存储平台来使用，总是需要有一个可以往 Kafka 写入数据的生产者和一个可以从 Kafka 读取数据的消费者，或者一个兼具两种角色的应用程序。

例如，在一个信用卡事务处理系统里，有一个客户端应用程序，它可能是一个在线商店，每当有支付行为发生时，它负责把事务发送到 Kafka 上。另一个应用程序根据规则引擎检查这个事务，决定是批准还是拒绝。批准或拒绝的响应消息被写回 Kafka，然后发送给发起事务的在线商店。第三个应用程序从 Kafka 上读取事务和审核状态，把它们保存到数据库，随后分析师可以对这些结果进行分析，或许还能借此改进规则引擎。

开发者们可以使用 Kafka 内置的客户端 API 开发 Kafka 应用程序。

在这一章，我们将从 Kafka 生产者的设计和组件讲起，学习如何使用 Kafka 生产者。我们将演示如何创建 KafkaProducer 和 ProducerRecords 对象、如何将记录发送给 Kafka，以及如何处理从 Kafka 返回的错误，然后介绍用于控制生产者行为的重要配置选项，最后深入探讨如何使用不同的分区方法和序列化器，以及如何自定义序列化器和分区器。

在第 4 章，我们将会介绍 Kafka 的消费者客户端，以及如何从 Kafka 读取消息。

> 第三方客户端
>
> 除了内置的客户端外，Kafka 还提供了二进制连接协议，也就是说，我们直接向 Kafka 网络端口发送适当的字节序列，就可以实现从 Kafka 读取消息或往 Kafka 写入消息。还有很多用其他语言实现的 Kafka 客户端，比如 C＋＋、 Python、Go 语言等，它们都实现了 Kafka 的连接协议，使得 Kafka 不仅仅局限于在 Java 里使用。这些客户端不属于 Kafka 项目，不过 Kafka 项目 wiki 上提供了一个清单，列出了所有可用的客户端。连接协议和第三方客户端超出了本章的讨论范围。

3.1　生产者概览

一个应用程序在很多情况下需要往 Kafka 写入消息：记录用户的活动（用于审计和分析）、记录度量指标、保存日志消息、记录智能家电的信息、与其他应用程序进行异步通信、缓冲即将写入到数据库的数据，等等。

多样的使用场景意味着多样的需求：是否每个消息都很重要？是否允许丢失一小部分消息？偶尔出现重复消息是否可以接受？是否有严格的延迟和吞吐量要求？

在之前提到的信用卡事务处理系统里，消息丢失或消息重复是不允许的，可以接受的延迟最大为 500ms，对吞吐量要求较高——我们希望每秒钟可以处理一百万个消息。

保存网站的点击信息是另一种使用场景。在这个场景里，允许丢失少量的消息或出现少量的消息重复，延迟可以高一些，只要不影响用户

体验就行。换句话说，只要用户点击链接后可以马上加载页面，那么我们并不介意消息要在几秒钟之后才能到达 Kafka 服务器。吞吐量则取决于网站用户使用网站的频度。

不同的使用场景对生产者 API 的使用和配置会有直接的影响。

尽管生产者 API 使用起来很简单，但消息的发送过程还是有点复杂的。图 3-1 展示了向 Kafka 发送消息的主要步骤。

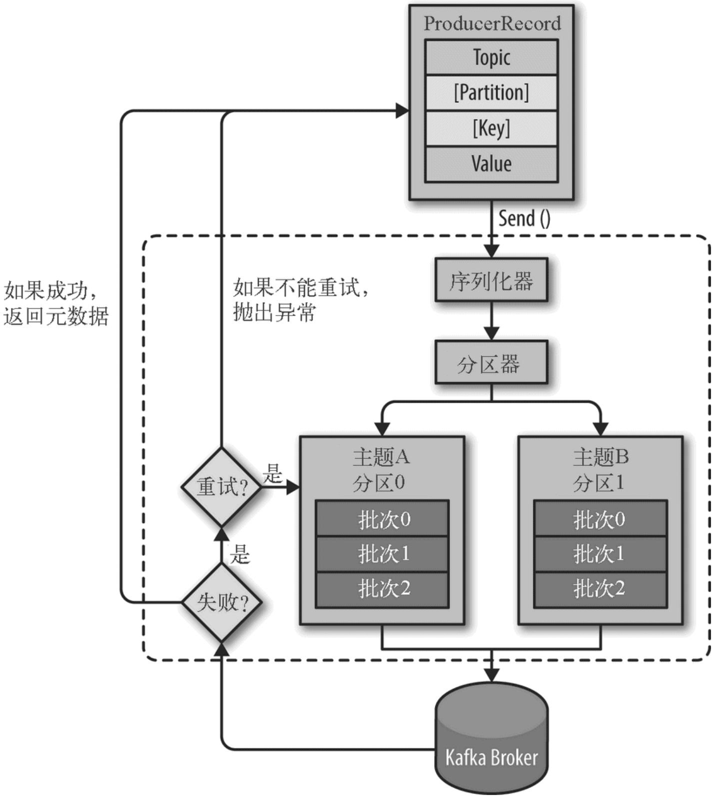


图 **3-1**：**Kafka** 生产者组件图

我们从创建一个 ProducerRecord 对象开始，ProducerRecord 对象需要包含目标主题和要发送的内容。我们还可以指定键或分区。在发送 ProducerRecord 对象时，生产者要先把键和值对象序列化成字节数组，这样它们才能够在网络上传输。

接下来，数据被传给分区器。如果之前在 ProducerRecord 对象里指

定了分区，那么分区器就不会再做任何事情，直接把指定的分区返回。如果没有指定分区，那么分区器会根据 ProducerRecord 对象的键来选择一个分区。选好分区以后，生产者就知道该往哪个主题和分区发送这条记录了。紧接着，这条记录被添加到一个记录批次里，这个批次里的所有消息会被发送到相同的主题和分区上。有一个独立的线程负责把这些记录批次发送到相应的 broker 上。

服务器在收到这些消息时会返回一个响应。如果消息成功写入 Kafka，就返回一个 RecordMetaData 对象，它包含了主题和分区信息，以及记录在分区里的偏移量。如果写入失败，则会返回一个错误。生产者在收到错误之后会尝试重新发送消息，几次之后如果还是失败，就返回错误信息。

3.2　创建Kafka生产者

要往 Kafka 写入消息，首先要创建一个生产者对象，并设置一些属性。Kafka 生产者有 3 个必选的属性。

`bootstrap.servers`

　　该属性指定 broker 的地址清单，地址的格式为 `host:port`。清单里不需要包含所有的 broker 地址，生产者会从给定的 broker 里查找到其他 broker 的信息。不过建议至少要提供两个 broker 的信息，一旦其中一个宕机，生产者仍然能够连接到集群上。

`key.serializer`

　　broker 希望接收到的消息的键和值都是字节数组。生产者接口允许使用参数化类型，因此可以把 Java 对象作为键和值发送给 broker。这样的代码具有良好的可读性，不过生产者需要知道如何把这些 Java 对象转换成字节数组。`key.serializer` 必须被设置为一个实现了 `org.apache.kafka.common.serialization.Serializer` 接口的类，生产者会使用这个类把键对象序列化成字节数组。Kafka 客户端默认提供了 `ByteArraySerializer`（这个只做很少的事情）、`StringSerializer` 和 `IntegerSerializer`，因此，如果你只使用常见的几种 Java 对象类型，那么就没必要实现自己的序列化器。要注意，`key.serializer` 是必须设置的，就算你打算只发送值内容。

value.serializer

与 key.serializer 一样，value.serializer 指定的类会将值序列化。如果键和值都是字符串，可以使用与 key.serializer 一样的序列化器。如果键是整数类型而值是字符串，那么需要使用不同的序列化器。

下面的代码片段演示了如何创建一个新的生产者，这里只指定了必要的属性，其他使用默认设置。

```
private Properties kafkaProps = new Properties(); ❶
kafkaProps.put("bootstrap.servers", "broker1:9092,broker2:9092");

kafkaProps.put("key.serializer",
  "org.apache.kafka.common.serialization.StringSerializer"); ❷
kafkaProps.put("value.serializer",
  "org.apache.kafka.common.serialization.StringSerializer");

producer = new KafkaProducer<String, String>(kafkaProps); ❸
```

❶ 新建一个 Properties 对象。

❷ 因为我们打算把键和值定义成字符串类型，所以使用内置的 StringSerializer。

❸ 在这里我们创建了一个新的生产者对象，并为键和值设置了恰当的类型，然后把 Properties 对象传给它。

这个接口很简单，通过配置生产者的不同属性就可以很大程度地控制它的行为。Kafka 的文档涵盖了所有的配置参数，我们将在这一章的后面部分介绍其中几个比较重要的参数。

实例化生产者对象后，接下来就可以开始发送消息了。发送消息主要有以下 3 种方式。

发送并忘记（**fire-and-forget**）

我们把消息发送给服务器，但并不关心它是否正常到达。大多数情况下，消息会正常到达，因为 Kafka 是高可用的，而且生产者会自动尝试重发。不过，使用这种方式有时候也会丢失一些消息。

同步发送

我们使用 send() 方法发送消息，它会返回一个 Future 对象，调用 get() 方法进行等待，就可以知道消息是否发送成功。

异步发送

我们调用 send() 方法，并指定一个回调函数，服务器在返回响应时调用该函数。

在下面的几个例子中，我们会介绍如何使用上述几种方式来发送消息，以及如何处理可能发生的异常情况。

本章的所有例子都使用单线程，但其实生产者是可以使用多线程来发送消息的。刚开始的时候可以使用单个消费者和单个线程。如果需要更高的吞吐量，可以在生产者数量不变的前提下增加线程数量。如果这样做还不够，可以增加生产者数量。

3.3 发送消息到Kafka

最简单的消息发送方式如下所示。

```
ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Precision Products",
          "France"); ❶
try {
  producer.send(record); ❷
} catch (Exception e) {
        e.printStackTrace(); ❸
}
```

❶ 生产者的 send() 方法将 ProducerRecord 对象作为参数，所以我们要先创建一个 ProducerRecord 对象。ProducerRecord 有多个构造函数，稍后我们会详细讨论。这里使用其中一个构造函数，它需要目标主题的名字和要发送的键和值对象，它们都是字符串。键和值对象的类型必须与序列化器和生产者对象相匹配。

❷ 我们使用生产者的 send() 方法发送 ProducerRecord 对象。从生产者的架构图里可以看到，消息先是被放进缓冲区，然后使用单独的线程发送到服务器端。send() 方法会返回一个包含 RecordMetadata 的 Future 对象，不过因为我们会忽略返回值，所以无法知道消息是否发送成功。如果不关心发送结果，那么可以使用这种发送方式。比如，记录 Twitter 消息日志，或记录不太重要的

应用程序日志。

❸ 我们可以忽略发送消息时可能发生的错误或在服务器端可能发生的错误，但在发送消息之前，生产者还是有可能发生其他的异常。这些异常有可能是 SerializationException（说明序列化消息失败）、BufferExhaustedException 或 TimeoutException（说明缓冲区已满），又或者是 InterruptException（说明发送线程被中断）。

3.3.1　同步发送消息

最简单的同步发送消息方式如下所示。

```
ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Precision Products", "Fra
try {
        producer.send(record).get(); ❶
} catch (Exception e) {
        e.printStackTrace(); ❷
}
```

❶ 在这里，`producer.send()` 方法先返回一个 Future 对象，然后调用 Future 对象的 `get()` 方法等待 Kafka 响应。如果服务器返回错误，`get()` 方法会抛出异常。如果没有发生错误，我们会得到一个 `RecordMetadata` 对象，可以用它获取消息的偏移量。

❷ 如果在发送数据之前或者在发送过程中发生了任何错误，比如 broker 返回了一个不允许重发消息的异常或者已经超过了重发的次数，那么就会抛出异常。我们只是简单地把异常信息打印出来。

`KafkaProducer` 一般会发生两类错误。其中一类是可重试错误，这类错误可以通过重发消息来解决。比如对于连接错误，可以通过再次建立连接来解决，“无主（no leader）”错误则可以通过重新为分区选举首领来解决。`KafkaProducer` 可以被配置成自动重试，如果在多次重试后仍无法解决问题，应用程序会收到一个重试异常。另一类错误无法通过重试解决，比如“消息太大”异常。对于这类错误，`KafkaProducer` 不会进行任何重试，直接抛出异常。

3.3.2　异步发送消息

假设消息在应用程序和 Kafka 集群之间一个来回需要 10ms。如果在发送完每个消息后都等待回应，那么发送 100 个消息需要 1 秒。但

如果只发送消息而不等待响应，那么发送 100 个消息所需要的时间会少很多。大多数时候，我们并不需要等待响应——尽管 Kafka 会把目标主题、分区信息和消息的偏移量发送回来，但对于发送端的应用程序来说不是必需的。不过在遇到消息发送失败时，我们需要抛出异常、记录错误日志，或者把消息写入“错误消息”文件以便日后分析。

为了在异步发送消息的同时能够对异常情况进行处理，生产者提供了回调支持。下面是使用回调的一个例子。

```
private class DemoProducerCallback implements Callback {❶
        @Override
    public void onCompletion(RecordMetadata recordMetadata, Exception e)
     if (e != null) {
         e.printStackTrace(); ❷
        }
    }
}

ProducerRecord<String, String> record =
        new ProducerRecord<>("CustomerCountry", "Biomedical Materials", "U
producer.send(record, new DemoProducerCallback()); ❹
```

❶ 为了使用回调，需要一个实现了 `org.apache.kafka.clients.producer.Callback` 接口的类，这个接口只有一个 `onCompletion` 方法。

❷ 如果 Kafka 返回一个错误，`onCompletion` 方法会抛出一个非空（non null）异常。这里我们只是简单地把它打印出来，但是在生产环境应该有更好的处理方式。

❸ 记录与之前的一样。

❹ 在发送消息时传进去一个回调对象。

3.4　生产者的配置

到目前为止，我们只介绍了生产者的几个必要配置参数——`bootstrap.servers` API 以及序列化器。

生产者还有很多可配置的参数，在 Kafka 文档里都有说明，它们大部分都有合理的默认值，所以没有必要去修改它们。不过有几个参数在内存使用、性能和可靠性方面对生产者影响比较大，接下来我们会一

一说明。

01. **acks**

acks 参数指定了必须要有多少个分区副本收到消息，生产者才会认为消息写入是成功的。这个参数对消息丢失的可能性有重要影响。该参数有如下选项。

- 如果 acks=0，生产者在成功写入消息之前不会等待任何来自服务器的响应。也就是说，如果当中出现了问题，导致服务器没有收到消息，那么生产者就无从得知，消息也就丢失了。不过，因为生产者不需要等待服务器的响应，所以它可以以网络能够支持的最大速度发送消息，从而达到很高的吞吐量。
- 如果 acks=1，只要集群的首领节点收到消息，生产者就会收到一个来自服务器的成功响应。如果消息无法到达首领节点（比如首领节点崩溃，新的首领还没有被选举出来），生产者会收到一个错误响应，为了避免数据丢失，生产者会重发消息。不过，如果一个没有收到消息的节点成为新首领，消息还是会丢失。这个时候的吞吐量取决于使用的是同步发送还是异步发送。如果让发送客户端等待服务器的响应（通过调用 Future 对象的 get() 方法），显然会增加延迟（在网络上传输一个来回的延迟）。如果客户端使用回调，延迟问题就可以得到缓解，不过吞吐量还是会受发送中消息数量的限制（比如，生产者在收到服务器响应之前可以发送多少个消息）。
- 如果 acks=all，只有当所有参与复制的节点全部收到消息时，生产者才会收到一个来自服务器的成功响应。这种模式是最安全的，它可以保证不止一个服务器收到消息，就算有服务器发生崩溃，整个集群仍然可以运行（第 5 章将讨论更多的细节）。不过，它的延迟比 acks=1 时更高，因为我们要等待不只一个服务器节点接收消息。

05. **buffer.memory**

该参数用来设置生产者内存缓冲区的大小，生产者用它缓冲要发送到服务器的消息。如果应用程序发送消息的速度超过发送到服务器的速度，会导致生产者空间不足。这个时候，send() 方法调用要么被阻塞，要么抛出异常，取决于如何设

置 `block.on.buffer.full` 参数（在 0.9.0.0 版本里被替换成了 `max.block.ms`，表示在抛出异常之前可以阻塞一段时间）。

06. **compression.type**

默认情况下，消息发送时不会被压缩。该参数可以设置为 `snappy`、`gzip` 或 `lz4`，它指定了消息被发送给 broker 之前使用哪一种压缩算法进行压缩。snappy 压缩算法由 Google 发明，它占用较少的 CPU，却能提供较好的性能和相当可观的压缩比，如果比较关注性能和网络带宽，可以使用这种算法。gzip 压缩算法一般会占用较多的 CPU，但会提供更高的压缩比，所以如果网络带宽比较有限，可以使用这种算法。使用压缩可以降低网络传输开销和存储开销，而这往往是向 Kafka 发送消息的瓶颈所在。

07. **retries**

生产者从服务器收到的错误有可能是临时性的错误（比如分区找不到首领）。在这种情况下，`retries` 参数的值决定了生产者可以重发消息的次数，如果达到这个次数，生产者会放弃重试并返回错误。默认情况下，生产者会在每次重试之间等待 100ms，不过可以通过 `retry.backoff.ms` 参数来改变这个时间间隔。建议在设置重试次数和重试时间间隔之前，先测试一下恢复一个崩溃节点需要多少时间（比如所有分区选举出首领需要多长时间），让总的重试时间比 Kafka 集群从崩溃中恢复的时间长，否则生产者会过早地放弃重试。不过有些错误不是临时性错误，没办法通过重试来解决（比如“消息太大”错误）。一般情况下，因为生产者会自动进行重试，所以就没必要在代码逻辑里处理那些可重试的错误。你只需要处理那些不可重试的错误或重试次数超出上限的情况。

08. **batch.size**

当有多个消息需要被发送到同一个分区时，生产者会把它们放在同一个批次里。该参数指定了一个批次可以使用的内存大小，按照字节数计算（而不是消息个数）。当批次被填满，批次里的所有消息会被发送出去。不过生产者并不一定都会等到批次被填满才发送，半满的批次，甚至只包含一个消息的批次也有可能被发送。所以就算把批次大小设置得很大，也不会造成延迟，只是会占用更多的内存而已。但如果设置得太小，因

为生产者需要更频繁地发送消息，会增加一些额外的开销。

09. **linger.ms**

该参数指定了生产者在发送批次之前等待更多消息加入批次的时间。`KafkaProducer` 会在批次填满或 `linger.ms` 达到上限时把批次发送出去。默认情况下，只要有可用的线程，生产者就会把消息发送出去，就算批次里只有一个消息。把 `linger.ms` 设置成比 0 大的数，让生产者在发送批次之前等待一会儿，使更多的消息加入到这个批次。虽然这样会增加延迟，但也会提升吞吐量（因为一次性发送更多的消息，每个消息的开销就变小了）。

10. **client.id**

该参数可以是任意的字符串，服务器会用它来识别消息的来源，还可以用在日志和配额指标里。

11. **max.in.flight.requests.per.connection**

该参数指定了生产者在收到服务器响应之前可以发送多少个消息。它的值越高，就会占用越多的内存，不过也会提升吞吐量。把它设为 1 可以保证消息是按照发送的顺序写入服务器的，即使发生了重试。

12. **timeout.ms**、**request.timeout.ms** 和 **metadata.fetch.timeout.ms**

`request.timeout.ms` 指定了生产者在发送数据时等待服务器返回响应的时间，`metadata.fetch.timeout.ms` 指定了生产者在获取元数据（比如目标分区的首领是谁）时等待服务器返回响应的时间。如果等待响应超时，那么生产者要么重试发送数据，要么返回一个错误（抛出异常或执行回调）。`timeout.ms` 指定了 broker 等待同步副本返回消息确认的时间，与 `asks` 的配置相匹配——如果在指定时间内没有收到同步副本的确认，那么 broker 就会返回一个错误。

13. **max.block.ms**

该参数指定了在调用 `send()` 方法或使用 `partitionsFor()` 方法获取元数据时生产者的阻塞时间。当生产者的发送缓冲区已满，或者没有可用的元数据时，这些方

法就会阻塞。在阻塞时间达到 `max.block.ms` 时，生产者会抛出超时异常。

14. **max.request.size**

    该参数用于控制生产者发送的请求大小。它可以指能发送的单个消息的最大值，也可以指单个请求里所有消息总的大小。例如，假设这个值为 1MB，那么可以发送的单个最大消息为 1MB，或者生产者可以在单个请求里发送一个批次，该批次包含了 1000 个消息，每个消息大小为 1KB。另外，broker 对可接收的消息最大值也有自己的限制（`message.max.bytes`），所以两边的配置最好可以匹配，避免生产者发送的消息被 broker 拒绝。

15. **receive.buffer.bytes** 和 **send.buffer.bytes**

    这两个参数分别指定了 TCP socket 接收和发送数据包的缓冲区大小。如果它们被设为 -1，就使用操作系统的默认值。如果生产者或消费者与 broker 处于不同的数据中心，那么可以适当增大这些值，因为跨数据中心的网络一般都有比较高的延迟和比较低的带宽。

> 顺序保证
>
> Kafka 可以保证同一个分区里的消息是有序的。也就是说，如果生产者按照一定的顺序发送消息，broker 就会按照这个顺序把它们写入分区，消费者也会按照同样的顺序读取它们。在某些情况下，顺序是非常重要的。例如，往一个账户存入 100 元再取出来，这个与先取钱再存钱是截然不同的！不过，有些场景对顺序不是很敏感。
>
> 如果把 `retries` 设为非零整数，同时把 `max.in.flight.requests.per.connection` 设为比 1 大的数，那么，如果第一个批次消息写入失败，而第二个批次写入成功，broker 会重试写入第一个批次。如果此时第一个批次也写入成功，那么两个批次的顺序就反过来了。
>
> 一般来说，如果某些场景要求消息是有序的，那么消息是否写入成功也是很关键的，所以不建议把

`retries` 设为 0。可以把 `max.in.flight.requests.per.connection` 设为 1，这样在生产者尝试发送第一批消息时，就不会有其他的消息发送给 broker。不过这样会严重影响生产者的吞吐量，所以只有在对消息的顺序有严格要求的情况下才能这么做。

3.5　序列化器

我们已经在之前的例子里看到，创建一个生产者对象必须指定序列化器。我们已经知道如何使用默认的字符串序列化器，Kafka 还提供了整型和字节数组序列化器，不过它们还不足以满足大部分场景的需求。到最后，我们需要序列化的记录类型会越来越多。

接下来演示如何开发自己的序列化器，并介绍 Avro 序列化器作为推荐的备选方案。

3.5.1　自定义序列化器

如果发送到 Kafka 的对象不是简单的字符串或整型，那么可以使用序列化框架来创建消息记录，如 Avro、Thrift 或 Protobuf，或者使用自定义序列化器。我们强烈建议使用通用的序列化框架。不过，为了了解序列化器的工作原理，也为了说明为什么要使用序列化框架，让我们一起来看看如何自定义一个序列化器。

假设你创建了一个简单的类来表示一个客户：

```
public class Customer {
        private int customerID;
        private String customerName;

        public Customer(int ID, String name) {
                this.customerID = ID;
                this.customerName = name;
        }

  public int getID() {
    return customerID;
  }

  public String getName() {
    return customerName;
  }
}
```

现在我们要为这个类创建一个序列化器，它看起来可能是这样的：

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerSerializer implements Serializer<Customer> {

        @Override
  public void configure(Map configs, boolean isKey) {
    // 不做任何配置
  }

  @Override
  /**
  Customer对象被序列化成：
  表示customerID的4字节整数
  表示customerName长度的4字节整数（如果customerName为空，则长度为0）
  表示customerName的N个字节
  */
  public byte[] serialize(String topic, Customer data) {
   try {
                  byte[] serializedName;
                  int stringSize;
      if (data == null)
        return null;
        else {
                                  if (data.getName() != null) {
           serializedName = data.getName().getBytes("UTF-8");
           stringSize = serializedName.length;
                                  } else {
                                          serializedName = new byte[0];
                                          stringSize = 0;
                                  }
                    }
        ByteBuffer buffer = ByteBuffer.allocate(4 + 4 + stringSize);
        buffer.putInt(data.getID());
        buffer.putInt(stringSize);
        buffer.put(serializedName);

        return buffer.array();
            } catch (Exception e) {
        throw new SerializationException("Error when serializing Customer
  byte[] " + e);
      }
    }
  @Override
  public void close() {
          // 不需要关闭任何东西
```

```
  }
}
```

只要使用这个 `CustomerSerializer`，就可以把消息记录定义成 `ProducerRecord<String, Customer>`，并且可以直接把 `Customer` 对象传给生产者。这个例子很简单，不过代码看起来太脆弱了——如果我们有多种类型的消费者，可能需要把 `customerID` 字段变成长整型，或者为 Customer 添加 `startDate` 字段，这样就会出现新旧消息的兼容性问题。在不同版本的序列化器和反序列化器之间调试兼容性问题着实是个挑战——你需要比较原始的字节数组。更糟糕的是，如果同一个公司的不同团队都需要往 Kafka 写入 Customer 数据，那么他们就需要使用相同的序列化器，如果序列化器发生改动，他们几乎要在同一时间修改代码。

基于以上几点原因，我们不建议使用自定义序列化器，而是使用已有的序列化器和反序列化器，比如 JSON、Avro、Thrift 或 Protobuf。下面我们将会介绍 Avro，然后演示如何序列化 Avro 记录并发送给 Kafka。

3.5.2　使用Avro序列化

Apache Avro（以下简称 Avro）是一种与编程语言无关的序列化格式。Doug Cutting 创建了这个项目，目的是提供一种共享数据文件的方式。

Avro 数据通过与语言无关的 schema 来定义。schema 通过 JSON 来描述，数据被序列化成二进制文件或 JSON 文件，不过一般会使用二进制文件。Avro 在读写文件时需要用到 schema，schema 一般会被内嵌在数据文件里。

Avro 有一个很有意思的特性是，当负责写消息的应用程序使用了新的 schema，负责读消息的应用程序可以继续处理消息而无需做任何改动，这个特性使得它特别适合用在像 Kafka 这样的消息系统上。

假设最初的 schema 是这样的：

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
     {"name": "id", "type": "int"},
     {"name": "name",  "type": "string"},
```

```
    {"name": "faxNumber", "type": ["null", "string"], "default": "null"}
 ]
}
```

❶ id 和 name 字段是必需的，faxNumber 是可选的，默认为 null。

假设我们已经使用了这个 schema 几个月的时间，并用它生成了几个太字节的数据。现在，我们决定在新版本里做一些修改。因为在 21 世纪不再需要 faxNumber 字段，需要用 email 字段来代替它。

新的 schema 如下：

```
{"namespace": "customerManagement.avro",
 "type": "record",
 "name": "Customer",
 "fields": [
    {"name": "id", "type": "int"},
    {"name": "name",  "type": "string"},
    {"name": "email", "type": ["null", "string"], "default": "null"}
 ]
}
```

更新到新版的 schema 后，旧记录仍然包含 faxNumber 字段，而新记录则包含 email 字段。部分负责读取数据的应用程序进行了升级，那么它们是如何处理这些变化的呢？

在应用程序升级之前，它们会调用类似 getName()、getId() 和 getFaxNumber() 这样的方法。如果碰到使用新 schema 构建的消息，getName() 和 getId() 方法仍然能够正常返回，但 getFaxNumber() 方法会返回 null，因为消息里不包含传真号码。

在应用程序升级之后，getEmail() 方法取代了 getFaxNumber() 方法。如果碰到一个使用旧 schema 构建的消息，那么 getEmail() 方法会返回 null，因为旧消息不包含邮件地址。

现在可以看出使用 Avro 的好处了：我们修改了消息的 schema，但并没有更新所有负责读取数据的应用程序，而这样仍然不会出现异常或阻断性错误，也不需要对现有数据进行大幅更新。

不过这里有以下两个需要注意的地方。

- 用于写入数据和读取数据的 schema 必须是相互兼容的。Avro 文档提到了一些兼容性原则。
- 反序列化器需要用到用于写入数据的 schema，即使它可能与用于读取数据的 schema 不一样。Avro 数据文件里就包含了用于写入数据的 schema，不过在 Kafka 里有一种更好的处理方式，下一小节我们会介绍它。

3.5.3　在Kafka里使用Avro

Avro 的数据文件里包含了整个 schema，不过这样的开销是可接受的。但是如果在每条 Kafka 记录里都嵌入 schema，会让记录的大小成倍地增加。不过不管怎样，在读取记录时仍然需要用到整个 schema，所以要先找到 schema。我们遵循通用的结构模式并使用“schema 注册表”来达到目的。schema 注册表并不属于 Kafka，现在已经有一些开源的 schema 注册表实现。在这个例子里，我们使用的是 Confluent Schema Registry。该注册表的代码可以在 GitHub 上找到，你也可以把它作为 Confluent 平台的一部分进行安装。如果你决定使用这个注册表，可以参考它的文档。

我们把所有写入数据需要用到的 schema 保存在注册表里，然后在记录里引用 schema 的标识符。负责读取数据的应用程序使用标识符从注册表里拉取 schema 来反序列化记录。序列化器和反序列化器分别负责处理 schema 的注册和拉取。Avro 序列化器的使用方法与其他序列化器是一样的。

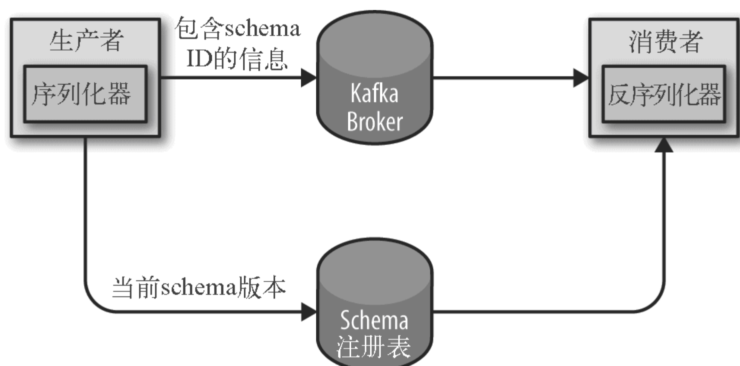


图 **3-2**：**Avro** 记录的序列化和反序列化流程图

下面的例子演示了如何把生成的 Avro 对象发送到 Kafka（关于如何使用 Avro 生成代码请参考 Avro 文档）：

```
Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("value.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
props.put("schema.registry.url", schemaUrl); ❷

String topic = "customerContacts";

Producer<String, Customer> producer = new KafkaProducer<String,
   Customer>(props); ❸

// 不断生成事件，直到有人按下Ctrl+C组合键
while (true) {
    Customer customer = CustomerGenerator.getNext();
    System.out.println("Generated customer " +
       customer.toString());
    ProducerRecord<String, Customer> record =
                        new ProducerRecord<>(topic, customer.getId(), cust
    producer.send(record); ❺
}
```

❶ 使用 Avro 的 `KafkaAvroSerializer` 来序列化对象。注意，AvroSerializer 也可以处理原语，这就是我们以后可以使用字符串作为记录键、使用客户对象作为值的原因。

❷ `schema.registry.url` 是一个新的参数，指向 schema 的存储位置。

❸ `Customer` 是生成的对象。我们会告诉生产者 `Customer` 对象就是记录的值。

❹ 实例化一个 `ProducerRecord` 对象，并指定 `Customer` 为值的类型，然后再传给它一个 `Customer` 对象。

❺ 把 `Customer` 对象作为记录发送出去，`KafkaAvroSerializer` 会处理剩下的事情。

如果你选择使用一般的 Avro 对象而非生成的 Avro 对象该怎么办？不用担心，这个时候你只需提供 schema 就可以了：

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("key.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer"); ❶
```

```
props.put("value.serializer",
   "io.confluent.kafka.serializers.KafkaAvroSerializer");
props.put("schema.registry.url", url); ❷

    String schemaString = "{\"namespace\": \"customerManagement.avro\",

\"type\": \"record\", " + ❸
                           "\"name\": \"Customer\"," +
                           "\"fields\": [" +
                           "{\"name\": \"id\", \"type\": \"int\"}," +
                           "{\"name\": \"name\", \"type\": \"string\"}," +
                           "{\"name\": \"email\", \"type\": [\"null\",\"st
\"], \"default\":\"null\" }" +
                           "]}";

    Producer<String, GenericRecord> producer =
       new KafkaProducer<String, GenericRecord>(props); ❹

    Schema.Parser parser = new Schema.Parser();
    Schema schema = parser.parse(schemaString);

    for (int nCustomers = 0; nCustomers < customers; nCustomers++) {
      String name = "exampleCustomer" + nCustomers;
      String email = "example" + nCustomers + "@example.com";

      GenericRecord customer = new GenericData.Record(schema); ❺
      customer.put("id", nCustomers);
      customer.put("name", name);
      customer.put("email", email);

      ProducerRecord<String, GenericRecord> data =
                               new ProducerRecord<String,
                                  GenericRecord>("customerContacts",
name, customer);
      producer.send(data);
   }
}
```

❶ 仍然使用同样的 `KafkaAvroSerializer`。

❷ 提供同样的 schema 注册表 URI。

❸ 这里需要提供 Avro schema，因为我们没有使用 Avro 生成的对象。

❹ 对象类型是 `Avro GenericRecord`，我们通过 schema 和需要写入的数据来初始化它。

❺ `ProducerRecord` 的值就是一个 `GenericRecord` 对象，它包

含了 schema 和数据。序列化器知道如何从记录里获取 schema，把它保存到注册表里，并用它序列化对象数据。

3.6　分区

在之前的例子里，ProducerRecord 对象包含了目标主题、键和值。Kafka 的消息是一个个键值对，ProducerRecord 对象可以只包含目标主题和值，键可以设置为默认的 null，不过大多数应用程序会用到键。键有两个用途：可以作为消息的附加信息，也可以用来决定消息该被写到主题的哪个分区。拥有相同键的消息将被写到同一个分区。也就是说，如果一个进程只从一个主题的分区读取数据（第 4 章会介绍更多细节），那么具有相同键的所有记录都会被该进程读取。要创建一个包含键值的记录，只需像下面这样创建 ProducerRecord 对象：

```
ProducerRecord<Integer, String> record =
        new ProducerRecord<>("CustomerCountry", "Laboratory Equipment", "U
```

如果要创建键为 null 的消息，不指定键就可以了：

```
ProducerRecord<Integer, String> record =
        new ProducerRecord<>("CustomerCountry", "USA"); ❶
```

❶ 这里的键被设为 null。

如果键值为 null，并且使用了默认的分区器，那么记录将被随机地发送到主题内各个可用的分区上。分区器使用轮询（Round Robin）算法将消息均衡地分布到各个分区上。

如果键不为空，并且使用了默认的分区器，那么 Kafka 会对键进行散列（使用 Kafka 自己的散列算法，即使升级 Java 版本，散列值也不会发生变化），然后根据散列值把消息映射到特定的分区上。这里的关键之处在于，同一个键总是被映射到同一个分区上，所以在进行映射时，我们会使用主题所有的分区，而不仅仅是可用的分区。这也意味着，如果写入数据的分区是不可用的，那么就会发生错误。但这种情况很少发生。我们将在第 6 章讨论 Kafka 的复制功能和可用性。

只有在不改变主题分区数量的情况下，键与分区之间的映射才能保持不变。举个例子，在分区数量保持不变的情况下，可以保证用户

045189 的记录总是被写到分区 34。在从分区读取数据时，可以进行各种优化。不过，一旦主题增加了新的分区，这些就无法保证了——旧数据仍然留在分区 34，但新的记录可能被写到其他分区上。如果要使用键来映射分区，那么最好在创建主题的时候就把分区规划好（第 2 章介绍了如何确定合适的分区数量），而且永远不要增加新分区。

实现自定义分区策略

我们已经讨论了默认分区器的特点，它是使用次数最多的分区器。不过，除了散列分区之外，有时候也需要对数据进行不一样的分区。假设你是一个 B2B 供应商，你有一个大客户，它是手持设备 Banana 的制造商。Banana 占据了你整体业务 10% 的份额。如果使用默认的散列分区算法，Banana 的账号记录将和其他账号记录一起被分配给相同的分区，导致这个分区比其他分区要大一些。服务器可能因此出现存储空间不足、处理缓慢等问题。我们需要给 Banana 分配单独的分区，然后使用散列分区算法处理其他账号。

下面是一个自定义分区器的例子：

```
import org.apache.kafka.clients.producer.Partitioner;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.common.PartitionInfo;
import org.apache.kafka.common.record.InvalidRecordException;
import org.apache.kafka.common.utils.Utils;

public class BananaPartitioner implements Partitioner {

        public void configure(Map<String, ?> configs) {} ❶

        public int partition(String topic, Object key, byte[] keyBytes,
                                   Object value, byte[] valueBytes,
                                     Cluster cluster) {
                List<PartitionInfo> partitions =
                  cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();

        if ((keyBytes == null) || (!(key instanceOf String))) ❷
          throw new InvalidRecordException("We expect all messages
            to have customer name as key")

        if (((String) key).equals("Banana"))
         return numPartitions; // Banana总是被分配到最后一个分区

        // 其他记录被散列到其他分区
        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1))
        }
```

```
    public void close() {}
}
```

❶ `Partitioner` 接口包含了 `configure`、`partition` 和 `close` 这 3 个方法。这里我们只实现 `partition` 方法，不过我们真不应该在 `partition` 方法里硬编码客户的名字，而应该通过 `configure` 方法传进来。

❷ 我们只接受字符串作为键，如果不是字符串，就抛出异常。

3.7　旧版的生产者API

在这一章，我们讨论了生产者的 Java 客户端，它是 `org.apache.kafka.clients` 包的一部分。在写到这一章的时候，Kafka 还有两个旧版的 Scala 客户端，它们是 `Kafka.producer` 包的一部分，同时也是 Kafka 的核心模块，它们是 SyncProducer（根据 `acks` 参数的具体配置情况，在发送更多的消息之前，它会等待服务器对已发消息或批次进行确认）和 AsyncProducer（在后台将消息分为不同的批次，使用单独的线程发送这些批次，不为客户端提供发送结果）。

因为当前版本的生产者 API 同时支持上述两种发送方式，而且为开发者提供了更高的可靠性和灵活性，所以我们不再讨论旧版的 API。如果你想使用它们，那么在使用之前请再三考虑，如果确定要使用，可以从 Kafka 文档中了解更多的信息。

3.8　总结

我们以一个生产者示例开始了本章的内容——使用 10 行代码将消息发送到 Kafka。然后我们在代码中加入错误处理逻辑，并介绍了同步和异步两种发送方式。接下来，我们介绍了生产者的一些重要配置参数以及它们对生产者行为的影响。我们还讨论了用于控制消息格式的序列化器，并深入探讨了 Avro——一种在 Kafka 中得到广泛应用的序列化方式。最后，我们讨论了 Kafka 的分区机制，并给出了一个自定义分区的例子。

现在我们已经知道如何向 Kafka 写入消息，在第 4 章，我们将学习如何从 Kafka 读取消息。

第 4 章　Kafka 消费者——从 Kafka 读取数据

应用程序使用 `KafkaConsumer` 向 Kafka 订阅主题，并从订阅的主题上接收消息。从 Kafka 读取数据不同于从其他消息系统读取数据，它涉及一些独特的概念和想法。如果不先理解这些概念，就难以理解如何使用消费者 API。所以我们接下来先解释这些重要的概念，然后再举几个例子，演示如何使用消费者 API 实现不同的应用程序。

4.1　KafkaConsumer概念

要想知道如何从 Kafka 读取消息，需要先了解消费者和消费者群组的概念。以下章节将解释这些概念。

4.1.1　消费者和消费者群组

假设我们有一个应用程序需要从一个 Kafka 主题读取消息并验证这些消息，然后再把它们保存起来。应用程序需要创建一个消费者对象，订阅主题并开始接收消息，然后验证消息并保存结果。过了一阵子，生产者往主题写入消息的速度超过了应用程序验证数据的速度，这个时候该怎么办？如果只使用单个消费者处理消息，应用程序会远跟不上消息生成的速度。显然，此时很有必要对消费者进行横向伸缩。就像多个生产者可以向相同的主题写入消息一样，我们也可以使用多个消费者从同一个主题读取消息，对消息进行分流。

Kafka 消费者从属于消费者群组。一个群组里的消费者订阅的是同一个主题，每个消费者接收主题一部分分区的消息。

假设主题 T1 有 4 个分区，我们创建了消费者 C1，它是群组 G1 里唯一的消费者，我们用它订阅主题 T1。消费者 C1 将收到主题 T1 全部 4 个分区的消息，如图 4-1 所示。

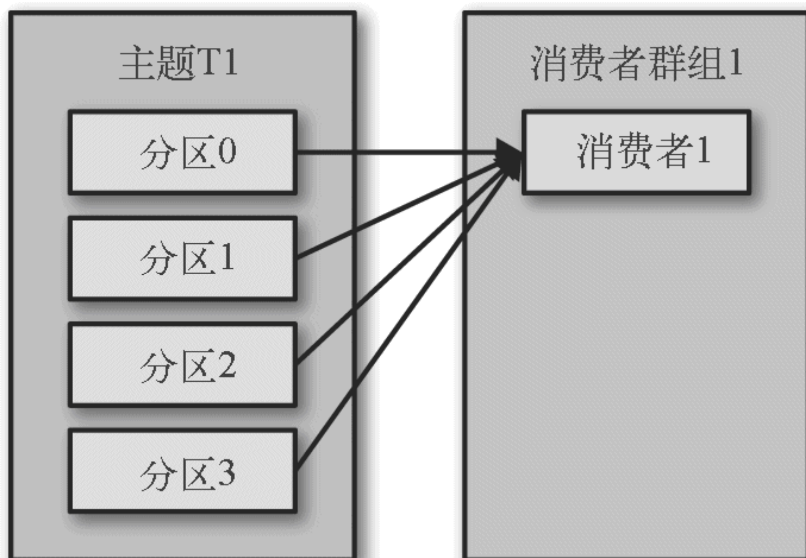


图 **4-1**：**1** 个消费者收到 **4** 个分区的消息

如果在群组 G1 里新增一个消费者 C2，那么每个消费者将分别从两个分区接收消息。我们假设消费者 C1 接收分区 0 和分区 2 的消息，消费者 C2 接收分区 1 和分区 3 的消息，如图 4-2 所示。

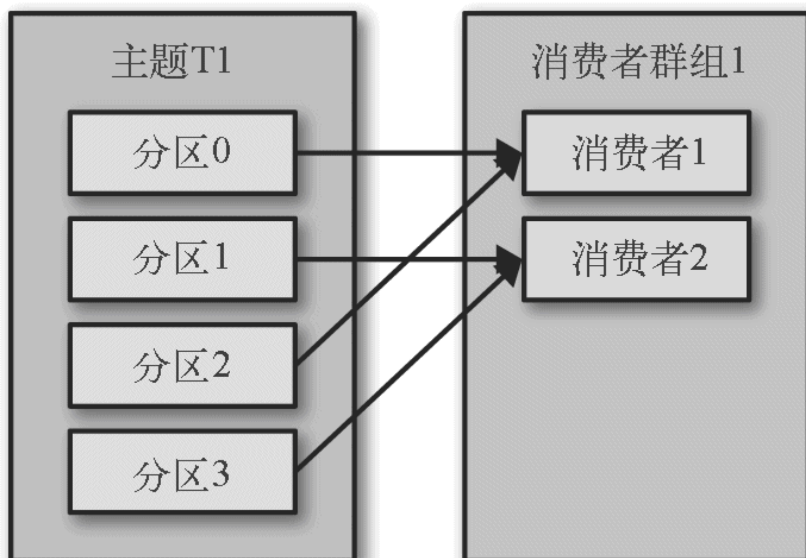


图 **4-2**：**2** 个消费者收到 **4** 个分区的消息

如果群组 G1 有 4 个消费者，那么每个消费者可以分配到一个分区，如图 4-3 所示。

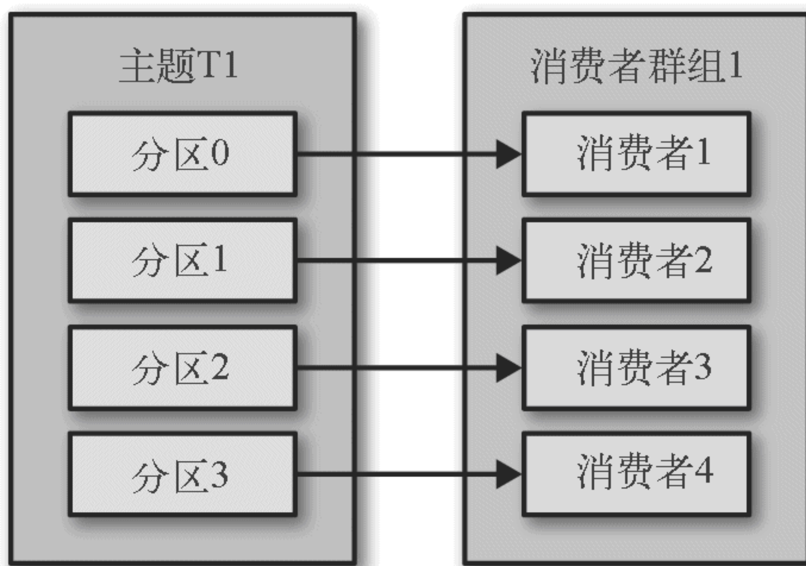


图 **4-3**：**4** 个消费者收到 **4** 个分区的消息

如果我们往群组里添加更多的消费者，超过主题的分区数量，那么有一部分消费者就会被闲置，不会接收到任何消息，如图 4-4 所示。

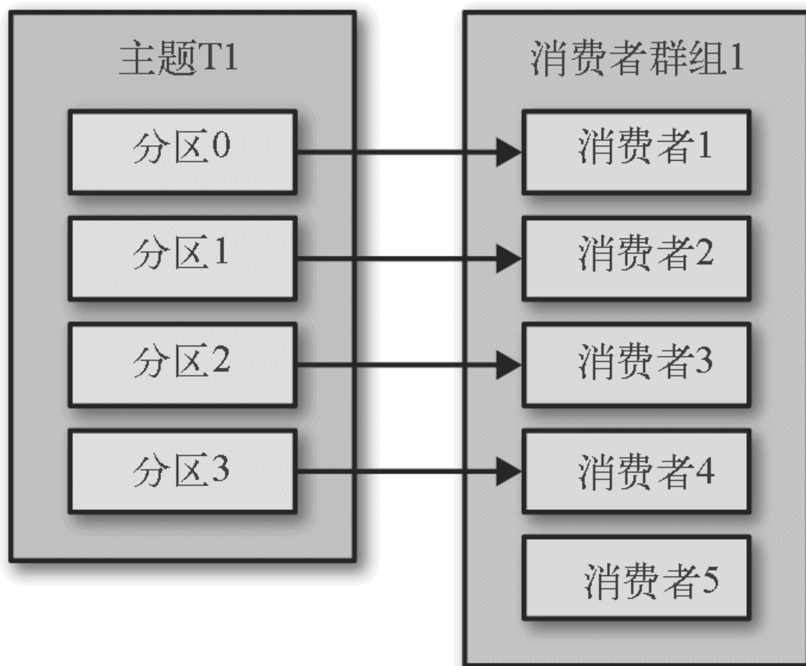


图 **4-4**：**5** 个消费者收到 **4** 个分区的消息

往群组里增加消费者是横向伸缩消费能力的主要方式。Kafka 消费者经常会做一些高延迟的操作，比如把数据写到数据库或 HDFS，或者使用数据进行比较耗时的计算。在这些情况下，单个消费者无法跟上数据生成的速度，所以可以增加更多的消费者，让它们分担负载，每个消费者只处理部分分区的消息，这就是横向伸缩的主要手段。我们有必要为主题创建大量的分区，在负载增长时可以加入更多的消费者。不过要注意，不要让消费者的数量超过主题分区的数量，多余的消费者只会被闲置。第 2 章介绍了如何为主题选择合适的分区数量。

除了通过增加消费者来横向伸缩单个应用程序外，还经常出现多个应用程序从同一个主题读取数据的情况。实际上，Kafka 设计的主要目标之一，就是要让 Kafka 主题里的数据能够满足企业各种应用场景的需求。在这些场景里，每个应用程序可以获取到所有的消息，而不只是其中的一部分。只要保证每个应用程序有自己的消费者群组，就可以让它们获取到主题所有的消息。不同于传统的消息系统，横向伸缩 Kafka 消费者和消费者群组并不会对性能造成负面影响。

在上面的例子里，如果新增一个只包含一个消费者的群组 G2，那么

这个消费者将从主题 T1 上接收所有的消息，与群组 G1 之间互不影响。群组 G2 可以增加更多的消费者，每个消费者可以消费若干个分区，就像群组 G1 那样，如图 4-5 所示。总的来说，群组 G2 还是会接收到所有消息，不管有没有其他群组存在。

简而言之，为每一个需要获取一个或多个主题全部消息的应用程序创建一个消费者群组，然后往群组里添加消费者来伸缩读取能力和处理能力，群组里的每个消费者只处理一部分消息。

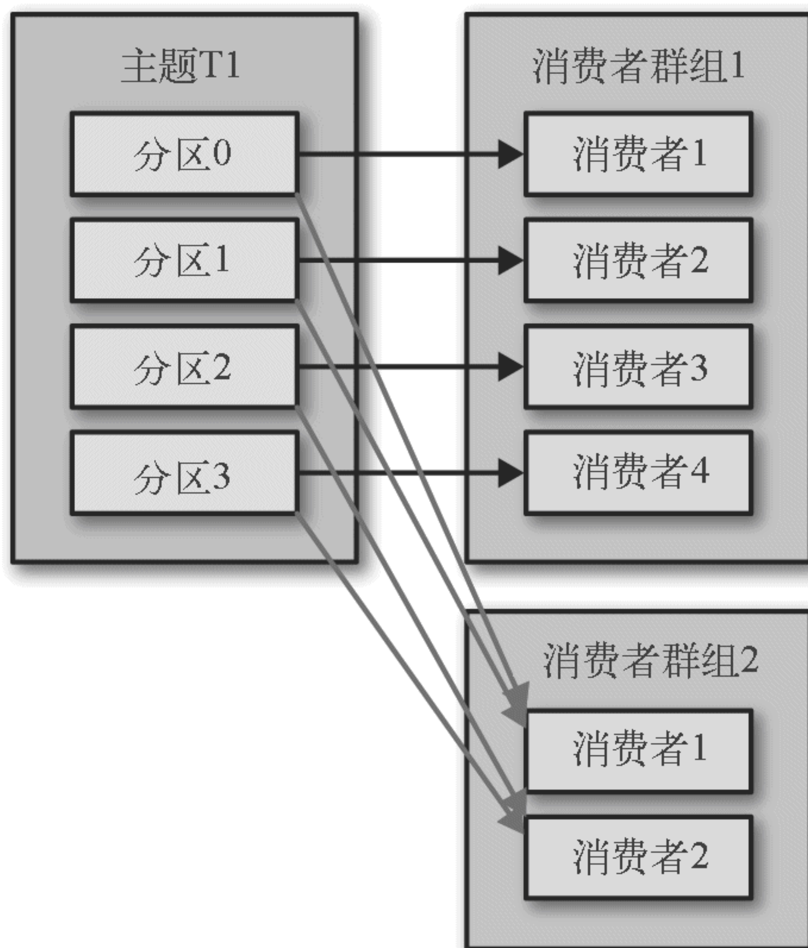


图 **4-5**：两个消费者群组对应一个主题

4.1.2　消费者群组和分区再均衡

我们已经从上一个小节了解到，群组里的消费者共同读取主题的分区。一个新的消费者加入群组时，它读取的是原本由其他消费者读取的消息。当一个消费者被关闭或发生崩溃时，它就离开群组，原本由它读取的分区将由群组里的其他消费者来读取。在主题发生变化时，比如管理员添加了新的分区，会发生分区重分配。

分区的所有权从一个消费者转移到另一个消费者，这样的行为被称为再均衡。再均衡非常重要，它为消费者群组带来了高可用性和伸缩性（我们可以放心地添加或移除消费者），不过在正常情况下，我们并不希望发生这样的行为。在再均衡期间，消费者无法读取消息，造成整个群组一小段时间的不可用。另外，当分区被重新分配给另一个消费者时，消费者当前的读取状态会丢失，它有可能还需要去刷新缓存，在它重新恢复状态之前会拖慢应用程序。我们将在本章讨论如何进行安全的再均衡，以及如何避免不必要的再均衡。

消费者通过向被指派为群组协调器的 broker（不同的群组可以有不同的协调器）发送心跳来维持它们和群组的从属关系以及它们对分区的所有权关系。只要消费者以正常的时间间隔发送心跳，就被认为是活跃的，说明它还在读取分区里的消息。消费者会在轮询消息（为了获取消息）或提交偏移量时发送心跳。如果消费者停止发送心跳的时间足够长，会话就会过期，群组协调器认为它已经死亡，就会触发一次再均衡。

如果一个消费者发生崩溃，并停止读取消息，群组协调器会等待几秒钟，确认它死亡了才会触发再均衡。在这几秒钟时间里，死掉的消费者不会读取分区里的消息。在清理消费者时，消费者会通知协调器它将要离开群组，协调器会立即触发一次再均衡，尽量降低处理停顿。在本章的后续部分，我们将讨论一些用于控制发送心跳频率和会话过期时间的配置参数，以及如何根据实际需要来配置这些参数。

心跳行为在最近版本中的变化

在 0.10.1 版本里，Kafka 社区引入了一个独立的心跳线程，可以在轮询消息的空档发送心跳。这样一来，发送心跳的频率（也就是消费者群组用于检测发生崩溃的消费者或不再发送心跳的消费者的时间）与消息轮询的频率（由处理消息所花费的时间来确定）之间就是相互独立的。在新版本的 Kafka 里，可以指定消费者在离开群组并触发再均衡之前可以有多长时间不进行消息轮询，这样可以避免出现活锁（livelock），比如有时候应用程序并没有崩溃，只

是由于某些原因导致无法正常运行。这个配置与 `session.timeout.ms` 是相互独立的，后者用于控制检测消费者发生崩溃的时间和停止发送心跳的时间。

本章的剩余部分将会讨论使用旧版本 Kafka 会面临的一些问题，以及如何解决这些问题。本章还包括如何应对需要较长时间来处理消息的情况的讨论，这些与 0.10.1 或更高版本的 Kafka 没有太大关系。如果你使用的是较新版本的Kafka，并且需要处理耗费较长时间的消息，只需要加大 `max.poll.interval.ms` 的值来增加轮询间隔的时长。

分配分区是怎样的一个过程

当消费者要加入群组时，它会向群组协调器发送一个 JoinGroup 请求。第一个加入群组的消费者将成为“群主”。群主从协调器那里获得群组的成员列表（列表中包含了所有最近发送过心跳的消费者，它们被认为是活跃的），并负责给每一个消费者分配分区。它使用一个实现了 `PartitionAssignor` 接口的类来决定哪些分区应该被分配给哪个消费者。

Kafka 内置了两种分配策略，在后面的配置参数小节我们将深入讨论。分配完毕之后，群主把分配情况列表发送给群组协调器，协调器再把这些信息发送给所有消费者。每个消费者只能看到自己的分配信息，只有群主知道群组里所有消费者的分配信息。这个过程会在每次再均衡时重复发生。

4.2　创建Kafka消费者

在读取消息之前，需要先创建一个 `KafkaConsumer` 对象。创建 `KafkaConsumer` 对象与创建 `KafkaProducer` 对象非常相似——把想要传给消费者的属性放在 `Properties` 对象里。本章后续部分会深入讨论所有的属性。在这里，我们只需要使用 3 个必要的属性：`bootstrap.servers`、`key.deserializer` 和

value.deserializer。

第 1 个属性 bootstrap.servers 指定了 Kafka 集群的连接字符串。它的用途与在 KafkaProducer 中的用途是一样的，可以参考第 3 章了解它的详细定义。另外两个属性 key.deserializer 和 value.deserializer 与生产者的 serializer 定义也很类似，不过它们不是使用指定的类把 Java 对象转成字节数组，而是使用指定的类把字节数组转成 Java 对象。

第 4 个属性 group.id 不是必需的，不过我们现在姑且认为它是必需的。它指定了 KafkaConsumer 属于哪一个消费者群组。创建不属于任何一个群组的消费者也是可以的，只是这样做不太常见，在本书的大部分章节，我们都假设消费者是属于某个群组的。

下面的代码片段演示了如何创建一个 KafkaConsumer 对象：

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
  "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
  "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<String,
String>(props);
```

如果在第 3 章看过如何创建生产者，就应该很熟悉上面的这段代码。我们假设消费的键和值都是字符串类型，所以使用的是内置的 StringDeserializer，并且使用字符串类型创建了 KafkaConsumer 对象。唯一不同的是新增了 group.id 属性，它指定了消费者所属群组的名字。

4.3　订阅主题

创建好消费者之后，下一步可以开始订阅主题了。subscribe() 方法接受一个主题列表作为参数，使用起来很简单：

```
consumer.subscribe(Collections.singletonList("customerCountries")); ❶
```

❶ 为了简单起见，我们创建了一个只包含单个元素的列表，主题的名

字叫作“customer Countries”。

我们也可以在调用 `subscribe()` 方法时传入一个正则表达式。正则表达式可以匹配多个主题，如果有人创建了新的主题，并且主题的名字与正则表达式匹配，那么会立即触发一次再均衡，消费者就可以读取新添加的主题。如果应用程序需要读取多个主题，并且可以处理不同类型的数据，那么这种订阅方式就很管用。在 Kafka 和其他系统之间复制数据时，使用正则表达式的方式订阅多个主题是很常见的做法。

要订阅所有与 test 相关的主题，可以这样做：

```
consumer.subscribe("test.*");
```

4.4　轮询

消息轮询是消费者 API 的核心，通过一个简单的轮询向服务器请求数据。一旦消费者订阅了主题，轮询就会处理所有的细节，包括群组协调、分区再均衡、发送心跳和获取数据，开发者只需要使用一组简单的 API 来处理从分区返回的数据。消费者代码的主要部分如下所示：

```
try {
  while (true) { ❶
      ConsumerRecords<String, String> records = consumer.poll(100); ❷
      for (ConsumerRecord<String, String> record : records) ❸
      {
          log.debug("topic = %s, partition = %s, offset = %d, customer = %
             country = %s\n",
             record.topic(), record.partition(), record.offset(),
             record.key(), record.value());

          int updatedCount = 1;
          if (custCountryMap.countainsValue(record.value())) {
              updatedCount = custCountryMap.get(record.value()) + 1;
          }
          custCountryMap.put(record.value(), updatedCount)

          JSONObject json = new JSONObject(custCountryMap);
          System.out.println(json.toString(4)) ❹
      }
  }
} finally {
  consumer.close(); ❺
}
```

❶ 这是一个无限循环。消费者实际上是一个长期运行的应用程序，它通过持续轮询向 Kafka 请求数据。稍后我们会介绍如何退出循环，并关闭消费者。

❷ 这一行代码非常重要。就像鲨鱼停止移动就会死掉一样，消费者必须持续对 Kafka 进行轮询，否则会被认为已经死亡，它的分区会被移交给群组里的其他消费者。传给 `poll()` 方法的参数是一个超时时间，用于控制 `poll()` 方法的阻塞时间（在消费者的缓冲区里没有可用数据时会发生阻塞）。如果该参数被设为 0，`poll()` 会立即返回，否则它会在指定的毫秒数内一直等待 broker 返回数据。

❸ `poll()` 方法返回一个记录列表。每条记录都包含了记录所属主题的信息、记录所在分区的信息、记录在分区里的偏移量，以及记录的键值对。我们一般会遍历这个列表，逐条处理这些记录。`poll()` 方法有一个超时参数，它指定了方法在多久之后可以返回，不管有没有可用的数据都要返回。超时时间的设置取决于应用程序对响应速度的要求，比如要在多长时间内把控制权归还给执行轮询的线程。

❹ 把结果保存起来或者对已有的记录进行更新，处理过程也随之结束。在这里，我们的目的是统计来自各个地方的客户数量，所以使用了一个散列表来保存结果，并以 JSON 的格式打印结果。在真实场景里，结果一般会被保存到数据存储系统里。

❺ 在退出应用程序之前使用 `close()` 方法关闭消费者。网络连接和 socket 也会随之关闭，并立即触发一次再均衡，而不是等待群组协调器发现它不再发送心跳并认定它已死亡，因为那样需要更长的时间，导致整个群组在一段时间内无法读取消息。

轮询不只是获取数据那么简单。在第一次调用新消费者的 `poll()` 方法时，它会负责查找 **GroupCoordinator**，然后加入群组，接受分配的分区。如果发生了再均衡，整个过程也是在轮询期间进行的。当然，心跳也是从轮询里发送出去的。所以，我们要确保在轮询期间所做的任何处理工作都应该尽快完成。

> 线程安全
>
> 在同一个群组里，我们无法让一个线程运行多个消费者，也无法让多个线程安全地共享一个消费者。按照规则，一个消费者使用一个线程。如果要在同一个消费者群组里运行多个消费者，需要让每个消费者运行在自己的线程里。最好是把消费者的逻辑封装在自己

的对象里，然后使用 Java 的 **ExecutorService** 启动多个线程，使每个消费者运行在自己的线程上。Confluent 的博客（https://www.confluent.io/blog/）上有一个教程介绍如何处理这种情况。

4.5 消费者的配置

到目前为止，我们学习了如何使用消费者 API，不过只介绍了几个配置属性——`bootstrap.servers`、`group.id`、`key.deserializer` 和 `value.deserializer`。Kafka 的文档列出了所有与消费者相关的配置说明。大部分参数都有合理的默认值，一般不需要修改它们，不过有一些参数与消费者的性能和可用性有很大关系。接下来介绍这些重要的属性。

01. **fetch.min.bytes**

    该属性指定了消费者从服务器获取记录的最小字节数。broker 在收到消费者的数据请求时，如果可用的数据量小于 `fetch.min.bytes` 指定的大小，那么它会等到有足够的可用数据时才把它返回给消费者。这样可以降低消费者和 broker 的工作负载，因为它们在主题不是很活跃的时候（或者一天里的低谷时段）就不需要来来回回地处理消息。如果没有很多可用数据，但消费者的 CPU 使用率却很高，那么就需要把该属性的值设得比默认值大。如果消费者的数量比较多，把该属性的值设置得大一点可以降低 broker 的工作负载。

02. **fetch.max.wait.ms**

    我们通过 `fetch.min.bytes` 告诉 Kafka，等到有足够的数据时才把它返回给消费者。而 `feth.max.wait.ms` 则用于指定 broker 的等待时间，默认是 500ms。如果没有足够的数据流入 Kafka，消费者获取最小数据量的要求就得不到满足，最终导致 500ms 的延迟。如果要降低潜在的延迟（为了满足 SLA），可以把该参数值设置得小一些。如果 `fetch.max.wait.ms` 被设为 100ms，并且 `fetch.min.bytes` 被设为 1MB，那么 Kafka 在收到消费者的请求后，要么返回 1MB 数据，要么在 100ms 后返回所有可用的数据，就看哪个条件先得到满足。

03. **max.partition.fetch.bytes**

该属性指定了服务器从每个分区里返回给消费者的最大字节数。它的默认值是 1MB，也就是说，`KafkaConsumer.poll()` 方法从每个分区里返回的记录最多不超过 `max.partition.fetch.bytes` 指定的字节。如果一个主题有 20 个分区和 5 个消费者，那么每个消费者需要至少 4MB 的可用内存来接收记录。在为消费者分配内存时，可以给它们多分配一些，因为如果群组里有消费者发生崩溃，剩下的消费者需要处理更多的分区。`max.partition.fetch.bytes` 的值必须比 broker 能够接收的最大消息的字节数（通过 `max.message.size` 属性配置）大，否则消费者可能无法读取这些消息，导致消费者一直挂起重试。在设置该属性时，另一个需要考虑的因素是消费者处理数据的时间。消费者需要频繁调用 `poll()` 方法来避免会话过期和发生分区再均衡，如果单次调用 `poll()` 返回的数据太多，消费者需要更多的时间来处理，可能无法及时进行下一个轮询来避免会话过期。如果出现这种情况，可以把 `max.partition.fetch.bytes` 值改小，或者延长会话过期时间。

04. **session.timeout.ms**

该属性指定了消费者在被认为死亡之前可以与服务器断开连接的时间，默认是 3s。如果消费者没有在 `session.timeout.ms` 指定的时间内发送心跳给群组协调器，就被认为已经死亡，协调器就会触发再均衡，把它的分区分配给群组里的其他消费者。该属性与 `heartbeat.interval.ms` 紧密相关。`heartbeat.interval.ms` 指定了 `poll()` 方法向协调器发送心跳的频率，`session.timeout.ms` 则指定了消费者可以多久不发送心跳。所以，一般需要同时修改这两个属性，`heartbeat.interval.ms` 必须比 `session.timeout.ms` 小，一般是 `session.timeout.ms` 的三分之一。如果 `session.timeout.ms` 是 3s，那么 `heartbeat.interval.ms` 应该是 1s。把 `session.timeout.ms` 值设得比默认值小，可以更快地检测和恢复崩溃的节点，不过长时间的轮询或垃圾收集可能导致非预期的再均衡。把该属性的值设置得大一些，可以减少意外的再均衡，不过检测节点崩溃需要更长的时间。

05. **auto.offset.reset**

该属性指定了消费者在读取一个没有偏移量的分区或者偏移量无效的情况下（因消费者长时间失效，包含偏移量的记录已经过时并被删除）该作何处理。它的默认值是 `latest`，意思是说，在偏移量无效的情况下，消费者将从最新的记录开始读取数据（在消费者启动之后生成的记录）。另一个值是 `earliest`，意思是说，在偏移量无效的情况下，消费者将从起始位置读取分区的记录。

06. **enable.auto.commit**

我们稍后将介绍几种不同的提交偏移量的方式。该属性指定了消费者是否自动提交偏移量，默认值是 `true`。为了尽量避免出现重复数据和数据丢失，可以把它设为 `false`，由自己控制何时提交偏移量。如果把它设为 `true`，还可以通过配置 `auto.commit.interval.ms` 属性来控制提交的频率。

07. **partition.assignment.strategy**

我们知道，分区会被分配给群组里的消费者。`PartitionAssignor` 根据给定的消费者和主题，决定哪些分区应该被分配给哪个消费者。Kafka 有两个默认的分配策略。

Range

　　该策略会把主题的若干个连续的分区分配给消费者。假设消费者 C1 和消费者 C2 同时订阅了主题 T1 和主题 T2，并且每个主题有 3 个分区。那么消费者 C1 有可能分配到这两个主题的分区 0 和分区 1，而消费者 C2 分配到这两个主题的分区 2。因为每个主题拥有奇数个分区，而分配是在主题内独立完成的，第一个消费者最后分配到比第二个消费者更多的分区。只要使用了 Range 策略，而且分区数量无法被消费者数量整除，就会出现这种情况。

RoundRobin

　　该策略把主题的所有分区逐个分配给消费者。如果使用 RoundRobin 策略来给消费者 C1 和消费者 C2 分配分区，那么消费者 C1 将分到主题 T1 的分区 0 和分区 2 以及主题 T2 的分区 1，消费者 C2 将分配到主题 T1 的分区 1 以及主题 T2 的分区 0 和分区 2。一般来说，如果所有消费者都订阅相同的主题（这种情况很常见），RoundRobin 策略会给所有消费者分配

相同数量的分区（或最多就差一个分区）。

可以通过设置 `partition.assignment.strategy` 来选择分区策略。默认使用的是 `org.apache.kafka.clients.consumer.RangeAssignor`，这个类实现了 Range 策略，不过也可以把它改成 `org.apache.kafka.clients.consumer.RoundRobinAssigno`
我们还可以使用自定义策略，在这种情况下，`partition.assignment.strategy` 属性的值就是自定义类的名字。

08. **client.id**

    该属性可以是任意字符串，broker 用它来标识从客户端发送过来的消息，通常被用在日志、度量指标和配额里。

09. **max.poll.records**

    该属性用于控制单次调用 `call()` 方法能够返回的记录数量，可以帮你控制在轮询里需要处理的数据量。

10. **receive.buffer.bytes** 和 **send.buffer.bytes**

    socket 在读写数据时用到的 TCP 缓冲区也可以设置大小。如果它们被设为 -1，就使用操作系统的默认值。如果生产者或消费者与 broker 处于不同的数据中心内，可以适当增大这些值，因为跨数据中心的网络一般都有比较高的延迟和比较低的带宽。

4.6　提交和偏移量

每次调用 `poll()` 方法，它总是返回由生产者写入 Kafka 但还没有被消费者读取过的记录，我们因此可以追踪到哪些记录是被群组里的哪个消费者读取的。之前已经讨论过，Kafka 不会像其他 JMS 队列那样需要得到消费者的确认，这是 Kafka 的一个独特之处。相反，消费者可以使用 Kafka 来追踪消息在分区里的位置（偏移量）。

我们把更新分区当前位置的操作叫作提交。

那么消费者是如何提交偏移量的呢？消费者往一个叫作 `_consumer_offset` 的特殊主题发送消息，消息里包含每个分区的偏移量。如果消费者一直处于运行状态，那么偏移量就没有什么用

处。不过，如果消费者发生崩溃或者有新的消费者加入群组，就会触发再均衡，完成再均衡之后，每个消费者可能分配到新的分区，而不是之前处理的那个。为了能够继续之前的工作，消费者需要读取每个分区最后一次提交的偏移量，然后从偏移量指定的地方继续处理。

如果提交的偏移量小于客户端处理的最后一个消息的偏移量，那么处于两个偏移量之间的消息就会被重复处理，如图 4-6 所示。

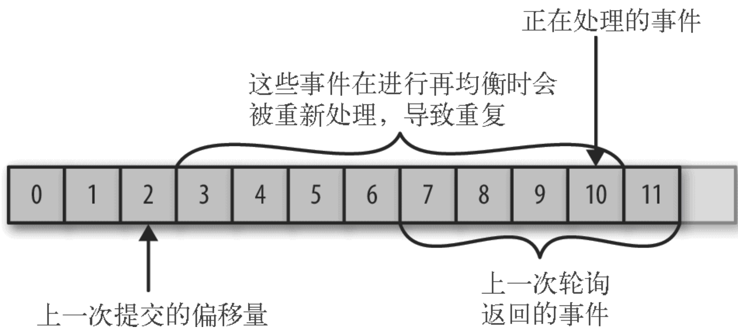


图 **4-6**：提交的偏移量小于客户端处理的最后一个消息的偏移量

如果提交的偏移量大于客户端处理的最后一个消息的偏移量，那么处于两个偏移量之间的消息将会丢失，如图 4-7 所示。

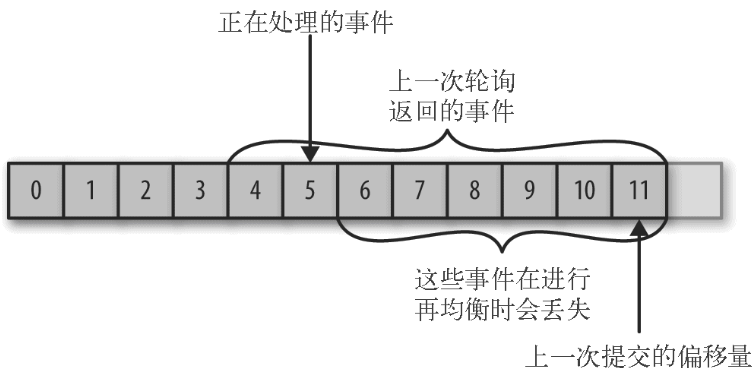


图 **4-7**：提交的偏移量大于客户端处理的最后一个消息的偏移量

所以，处理偏移量的方式对客户端会有很大的影响。

KafkaConsumer API 提供了很多种方式来提交偏移量。

4.6.1 自动提交

最简单的提交方式是让消费者自动提交偏移量。如果 enable.auto.commit 被设为 true，那么每过 5s，消费者会自动把从 poll() 方法接收到的最大偏移量提交上去。提交时间间隔由 auto.commit.interval.ms 控制，默认值是 5s。与消费者里的其他东西一样，自动提交也是在轮询里进行的。消费者每次在进行轮询时会检查是否该提交偏移量了，如果是，那么就会提交从上一次轮询返回的偏移量。

不过，在使用这种简便的方式之前，需要知道它将会带来怎样的结果。

假设我们仍然使用默认的 5s 提交时间间隔，在最近一次提交之后的 3s 发生了再均衡，再均衡之后，消费者从最后一次提交的偏移量位置开始读取消息。这个时候偏移量已经落后了 3s，所以在这 3s 内到达的消息会被重复处理。可以通过修改提交时间间隔来更频繁地提交偏移量，减小可能出现重复消息的时间窗，不过这种情况是无法完全避免的。

在使用自动提交时，每次调用轮询方法都会把上一次调用返回的偏移量提交上去，它并不知道具体哪些消息已经被处理了，所以在再次调用之前最好确保所有当前调用返回的消息都已经处理完毕（在调用 close() 方法之前也会进行自动提交）。一般情况下不会有什么问题，不过在处理异常或提前退出轮询时要格外小心。

自动提交虽然方便，不过并没有为开发者留有余地来避免重复处理消息。

4.6.2　提交当前偏移量

大部分开发者通过控制偏移量提交时间来消除丢失消息的可能性，并在发生再均衡时减少重复消息的数量。消费者 API 提供了另一种提交偏移量的方式，开发者可以在必要的时候提交当前偏移量，而不是基于时间间隔。

把 auto.commit.offset 设为 false，让应用程序决定何时提交偏移量。使用 commitSync() 提交偏移量最简单也最可靠。这个 API 会提交由 poll() 方法返回的最新偏移量，提交成功后马上返回，如果提交失败就抛出异常。

要记住，commitSync() 将会提交由 poll() 返回的最新偏移量，所以在处理完所有记录后要确保调用了 commitSync()，否则还是

会有丢失消息的风险。如果发生了再均衡，从最近一批消息到发生再均衡之间的所有消息都将被重复处理。

下面是我们在处理完最近一批消息后使用 commitSync() 方法提交偏移量的例子。

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset =
          %d, customer = %s, country = %s\n",
             record.topic(), record.partition(),
                  record.offset(), record.key(), record.value()); ❶
        }
        try {
          consumer.commitSync(); ❷
        } catch (CommitFailedException e) {
            log.error("commit failed", e) ❸
        }
}
```

❶ 我们假设把记录内容打印出来就算处理完毕，这个是由应用程序根据具体的使用场景来决定的。

❷ 处理完当前批次的消息，在轮询更多的消息之前，调用 commitSync() 方法提交当前批次最新的偏移量。

❸ 只要没有发生不可恢复的错误，commitSync() 方法会一直尝试直至提交成功。如果提交失败，我们也只能把异常记录到错误日志里。

4.6.3　异步提交

手动提交有一个不足之处，在 broker 对提交请求作出回应之前，应用程序会一直阻塞，这样会限制应用程序的吞吐量。我们可以通过降低提交频率来提升吞吐量，但如果发生了再均衡，会增加重复消息的数量。

这个时候可以使用异步提交 API。我们只管发送提交请求，无需等待 broker 的响应。

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
```

```
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s,
        offset = %d, customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    }
    consumer.commitAsync(); ❶
}
```

❶ 提交最后一个偏移量，然后继续做其他事情。

在成功提交或碰到无法恢复的错误之前，commitSync() 会一直重试，但是 commitAsync() 不会，这也是 commitAsync() 不好的一个地方。它之所以不进行重试，是因为在它收到服务器响应的时候，可能有一个更大的偏移量已经提交成功。假设我们发出一个请求用于提交偏移量 2000，这个时候发生了短暂的通信问题，服务器收不到请求，自然也不会作出任何响应。与此同时，我们处理了另外一批消息，并成功提交了偏移量 3000。如果 commitAsync() 重新尝试提交偏移量 2000，它有可能在偏移量 3000 之后提交成功。这个时候如果发生再均衡，就会出现重复消息。

我们之所以提到这个问题的复杂性和提交顺序的重要性，是因为 commitAsync() 也支持回调，在 broker 作出响应时会执行回调。回调经常被用于记录提交错误或生成度量指标，不过如果你要用它来进行重试，一定要注意提交的顺序。

```
while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records) {
        System.out.printf("topic = %s, partition = %s,
        offset = %d, customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value());
    }
    consumer.commitAsync(new OffsetCommitCallback() {
        public void onComplete(Map<TopicPartition,
        OffsetAndMetadata> offsets, Exception e) {
            if (e != null)
                log.error("Commit failed for offsets {}", offsets, e);
        }
      }); ❶
}
```

❶ 发送提交请求然后继续做其他事情，如果提交失败，错误信息和偏

移量会被记录下来。

> 重试异步提交
>
> 我们可以使用一个单调递增的序列号来维护异步提交的顺序。在每次提交偏移量之后或在回调里提交偏移量时递增序列号。在进行重试前，先检查回调的序列号和即将提交的偏移量是否相等，如果相等，说明没有新的提交，那么可以安全地进行重试。如果序列号比较大，说明有一个新的提交已经发送出去了，应该停止重试。

4.6.4 同步和异步组合提交

一般情况下，针对偶尔出现的提交失败，不进行重试不会有太大问题，因为如果提交失败是因为临时问题导致的，那么后续的提交总会有成功的。但如果这是发生在关闭消费者或再均衡前的最后一次提交，就要确保能够提交成功。

因此，在消费者关闭前一般会组合使用 `commitAsync()` 和 `commitSync()`。它们的工作原理如下（后面讲到再均衡监听器时，我们会讨论如何在发生再均衡前提交偏移量）：

```
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            System.out.println("topic = %s, partition = %s, offset = %d,
            customer = %s, country = %s\n",
            record.topic(), record.partition(),
            record.offset(), record.key(), record.value());
        }
        consumer.commitAsync(); ❶
    }
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(); ❷
    } finally {
        consumer.close();
    }
}
```

❶ 如果一切正常，我们使用 `commitAsync()` 方法来提交。这样速

度更快，而且即使这次提交失败，下一次提交很可能会成功。

❷ 如果直接关闭消费者，就没有所谓的“下一次提交”了。使用 `commitSync()` 方法会一直重试，直到提交成功或发生无法恢复的错误。

4.6.5 提交特定的偏移量

提交偏移量的频率与处理消息批次的频率是一样的。但如果想要更频繁地提交该怎么办？如果 `poll()` 方法返回一大批数据，为了避免因再均衡引起的重复处理整批消息，想要在批次中间提交偏移量该怎么办？这种情况无法通过调用 `commitSync()` 或 `commitAsync()` 来实现，因为它们只会提交最后一个偏移量，而此时该批次里的消息还没有处理完。

幸运的是，消费者 API 允许在调用 `commitSync()` 和 `commitAsync()` 方法时传进去希望提交的分区和偏移量的 map。假设你处理了半个批次的消息，最后一个来自主题“customers”分区 3 的消息的偏移量是 5000，你可以调用 `commitSync()` 方法来提交它。不过，因为消费者可能不只读取一个分区，你需要跟踪所有分区的偏移量，所以在这个层面上控制偏移量的提交会让代码变复杂。

下面是提交特定偏移量的例子：

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets =
    new HashMap<>(); ❶
int count = 0;

...

while (true) {
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        System.out.printf("topic = %s, partition = %s, offset = %d,
        customer = %s, country = %s\n",
        record.topic(), record.partition(), record.offset(),
        record.key(), record.value()); ❷
        currentOffsets.put(new TopicPartition(record.topic(),
        record.partition()), new
        OffsetAndMetadata(record.offset()+1, "no metadata")); ❸
        if (count % 1000 == 0) ❹
            consumer.commitAsync(currentOffsets,null); ❺
        count++;
    }
}
```

❶ 用于跟踪偏移量的 map。

❷ 记住，printf 只是处理消息的临时方案。

❸ 在读取每条记录之后，使用期望处理的下一个消息的偏移量更新 map 里的偏移量。下一次就从这里开始读取消息。

❹ 我们决定每处理 1000 条记录就提交一次偏移量。在实际应用中，你可以根据时间或记录的内容进行提交。

❺ 这里调用的是 commitAsync()，不过调用commitSync()也是完全可以的。当然，在提交特定偏移量时，仍然要处理可能发生的错误。

4.7 再均衡监听器

在提交偏移量一节中提到过，消费者在退出和进行分区再均衡之前，会做一些清理工作。

你会在消费者失去对一个分区的所有权之前提交最后一个已处理记录的偏移量。如果消费者准备了一个缓冲区用于处理偶发的事件，那么在失去分区所有权之前，需要处理在缓冲区累积下来的记录。你可能还需要关闭文件句柄、数据库连接等。

在为消费者分配新分区或移除旧分区时，可以通过消费者 API 执行一些应用程序代码，在调用 subscribe() 方法时传进去一个 ConsumerRebalanceListener 实例就可以了。ConsumerRebalanceListener 有两个需要实现的方法。

(1) public void onPartitionsRevoked(Collection<TopicPartition> partitions) 方法会在再均衡开始之前和消费者停止读取消息之后被调用。如果在这里提交偏移量，下一个接管分区的消费者就知道该从哪里开始读取了。

(2) public void onPartitionsAssigned(Collection<TopicPartition> partitions) 方法会在重新分配分区之后和消费者开始读取消息之前被调用。

下面的例子将演示如何在失去分区所有权之前通过 onPartitionsRevoked() 方法来提交偏移量。在下一节，我们会演示另一个同时使用了 onPartitionsAssigned() 方法的例子。

```
private Map<TopicPartition, OffsetAndMetadata> currentOffsets=
  new HashMap<>();

private class HandleRebalance implements ConsumerRebalanceListener { ❶
    public void onPartitionsAssigned(Collection<TopicPartition>
      partitions) { ❷
    }

    public void onPartitionsRevoked(Collection<TopicPartition>
      partitions) {
        System.out.println("Lost partitions in rebalance.
          Committing current
        offsets:" + currentOffsets);
        consumer.commitSync(currentOffsets); ❸
    }
}

try {
    consumer.subscribe(topics, new HandleRebalance()); ❹

    while (true) {
        ConsumerRecords<String, String> records =
          consumer.poll(100);
        for (ConsumerRecord<String, String> record : records)
        {
            System.out.println("topic = %s, partition = %s, offset = %d,
             customer = %s, country = %s\n",
             record.topic(), record.partition(), record.offset(),
             record.key(), record.value());
             currentOffsets.put(new TopicPartition(record.topic(),
             record.partition()), new
             OffsetAndMetadata(record.offset()+1, "no metadata"));
        }
        consumer.commitAsync(currentOffsets, null);
    }
} catch (WakeupException e) {
    // 忽略异常，正在关闭消费者
} catch (Exception e) {
    log.error("Unexpected error", e);
} finally {
    try {
        consumer.commitSync(currentOffsets);
    } finally {
        consumer.close();
        System.out.println("Closed consumer and we are done");
    }
}
```

❶ 首先实现 `ConsumerRebalanceListener` 接口。

❷ 在获得新分区后开始读取消息，不需要做其他事情。

❸ 如果发生再均衡，我们要在即将失去分区所有权时提交偏移量。要注意，提交的是最近处理过的偏移量，而不是批次中还在处理的最后一个偏移量。因为分区有可能在我们还在处理消息的时候被撤回。我们要提交所有分区的偏移量，而不只是那些即将失去所有权的分区的偏移量——因为提交的偏移量是已经处理过的，所以不会有什么问题。调用 `commitSync()` 方法，确保在再均衡发生之前提交偏移量。

❹ 把 `ConsumerRebalanceListener` 对象传给 `subscribe()` 方法，这是最重要的一步。

4.8　从特定偏移量处开始处理记录

到目前为止，我们知道了如何使用 `poll()` 方法从各个分区的最新偏移量处开始处理消息。不过，有时候我们也需要从特定的偏移量处开始读取消息。

如果你想从分区的起始位置开始读取消息，或者直接跳到分区的末尾开始读取消息，可以使用 `seekToBeginning(Collection<TopicPartition> tp)` 和 `seekToEnd(Collection<TopicPartition> tp)` 这两个方法。

不过，Kafka 也为我们提供了用于查找特定偏移量的 API。它有很多用途，比如向后回退几个消息或者向前跳过几个消息（对时间比较敏感的应用程序在处理滞后的情况下希望能够向前跳过若干个消息）。在使用 Kafka 以外的系统来存储偏移量时，它将给我们带来更大的惊喜。

试想一下这样的场景：应用程序从 Kafka 读取事件（可能是网站的用户点击事件流），对它们进行处理（可能是使用自动程序清理点击操作并添加会话信息），然后把结果保存到数据库、NoSQL 存储引擎或 Hadoop。假设我们真的不想丢失任何数据，也不想在数据库里多次保存相同的结果。

这种情况下，消费者的代码可能是这样的：

```
while (true) {
```

```
    ConsumerRecords<String, String> records = consumer.poll(100);
    for (ConsumerRecord<String, String> record : records)
    {
        currentOffsets.put(new TopicPartition(record.topic(),
        record.partition()),
        new OffsetAndMetadata(record.offset()+1);
        processRecord(record);
        storeRecordInDB(record);
        consumer.commitAsync(currentOffsets);
    }
}
```

在这个例子里，每处理一条记录就提交一次偏移量。尽管如此，在记录被保存到数据库之后以及偏移量被提交之前，应用程序仍然有可能发生崩溃，导致重复处理数据，数据库里就会出现重复记录。

如果保存记录和偏移量可以在一个原子操作里完成，就可以避免出现上述情况。记录和偏移量要么都被成功提交，要么都不提交。如果记录是保存在数据库里而偏移量是提交到 Kafka 上，那么就无法实现原子操作。

不过，如果在同一个事务里把记录和偏移量都写到数据库里会怎样呢？那么我们就会知道记录和偏移量要么都成功提交，要么都没有，然后重新处理记录。

现在的问题是：如果偏移量是保存在数据库里而不是 Kafka 里，那么消费者在得到新分区时怎么知道该从哪里开始读取？这个时候可以使用 `seek()` 方法。在消费者启动或分配到新分区时，可以使用 `seek()` 方法查找保存在数据库里的偏移量。

下面的例子大致说明了如何使用这个 API。使用 `ConsumerRebalanceListener` 和 `seek()` 方法确保我们是从数据库里保存的偏移量所指定的位置开始处理消息的。

```
public class SaveOffsetsOnRebalance implements
  ConsumerRebalanceListener {

    public void onPartitionsRevoked(Collection<TopicPartition>
      partitions) {
            commitDBTransaction(); ❶
      }

    public void onPartitionsAssigned(Collection<TopicPartition>
      partitions) {
        for(TopicPartition partition: partitions)
```

```
            consumer.seek(partition, getOffsetFromDB(partition)); ❷
        }
    }
}

  consumer.subscribe(topics, new SaveOffsetOnRebalance(consumer));
  consumer.poll(0);

  for (TopicPartition partition: consumer.assignment())
    consumer.seek(partition, getOffsetFromDB(partition)); ❸

  while (true) {
      ConsumerRecords<String, String> records =
        consumer.poll(100);
      for (ConsumerRecord<String, String> record : records)
      {
          processRecord(record);
          storeRecordInDB(record);
          storeOffsetInDB(record.topic(), record.partition(),
            record.offset()); ❹
      }
      commitDBTransaction();
}
```

❶ 使用一个虚构的方法来提交数据库事务。大致想法是这样的：在处理完记录之后，将记录和偏移量插入数据库，然后在即将失去分区所有权之前提交事务，确保成功保存了这些信息。

❷ 使用另一个虚构的方法来从数据库获取偏移量，在分配到新分区的时候，使用 `seek()` 方法定位到那些记录。

❸ 订阅主题之后，开始启动消费者，我们调用一次 `poll()` 方法，让消费者加入到消费者群组里，并获取分配到的分区，然后马上调用 `seek()` 方法定位分区的偏移量。要记住， `seek()` 方法只更新我们正在使用的位置，在下一次调用 `poll()` 时就可以获得正确的消息。如果 `seek()` 发生错误（比如偏移量不存在），`poll()` 就会抛出异常。

❹ 另一个虚构的方法，这次要更新的是数据库里用于保存偏移量的表。假设更新记录的速度非常快，所以每条记录都需要更新一次数据库，但提交的速度比较慢，所以只在每个批次末尾提交一次。这里可以通过很多种方式进行优化。

通过把偏移量和记录保存到同一个外部系统来实现单次语义可以有很多种方式，不过它们都需要结合使用 `ConsumerRebalanceListener` 和 `seek()` 方法来确保能够及时

保存偏移量，并保证消费者总是能够从正确的位置开始读取消息。

4.9 如何退出

在之前讨论轮询时就说过，不需要担心消费者会在一个无限循环里轮询消息，我们会告诉消费者如何优雅地退出循环。

如果确定要退出循环，需要通过另一个线程调用 consumer.wakeup() 方法。如果循环运行在主线程里，可以在 ShutdownHook 里调用该方法。要记住，consumer.wakeup() 是消费者唯一一个可以从其他线程里安全调用的方法。调用 consumer.wakeup() 可以退出 poll()，并抛出 WakeupException 异常，或者如果调用 consumer.wakeup() 时线程没有等待轮询，那么异常将在下一轮调用 poll() 时抛出。我们不需要处理 WakeupException，因为它只是用于跳出循环的一种方式。不过，在退出线程之前调用consumer.close()是很有必要的，它会提交任何还没有提交的东西，并向群组协调器发送消息，告知自己要离开群组，接下来就会触发再均衡，而不需要等待会话超时。

下面是运行在主线程上的消费者退出线程的代码。这些代码经过了简化，你可以在这里查看完整的代码：http://bit.ly/2u47e9A。

```
Runtime.getRuntime().addShutdownHook(new Thread() {
            public void run() {
                System.out.println("Starting exit...");
                consumer.wakeup(); ❶
                try {
                    mainThread.join();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });

...

try {
            // 循环，直到按下Ctrl+C键，关闭的钩子会在退出时进行清理
            while (true) {
                ConsumerRecords<String, String> records =
                  movingAvg.consumer.poll(1000);
                System.out.println(System.currentTimeMillis() + "
                  --  waiting for data...");
                for (ConsumerRecord<String, String> record :
```

```
                    records) {
                      System.out.printf("offset = %d, key = %s,
                        value = %s\n",
                        record.offset(), record.key(),
                        record.value());
                  }
                  for (TopicPartition tp: consumer.assignment())
                      System.out.println("Committing offset at
                        position:" +
                        consumer.position(tp));
                  movingAvg.consumer.commitSync();
              }
          } catch (WakeupException e) {
              // 忽略关闭异常 ❷
          } finally {
              consumer.close(); ❸
              System.out.println("Closed consumer and we are done");
          }
      }
```

❶ ShutdownHook 运行在单独的线程里，所以退出循环最安全的方式只能是调用 wakeup() 方法。

❷ 在另一个线程里调用 wakeup() 方法，导致 poll() 抛出 WakeupException。你可能想捕获异常以确保应用不会意外终止，但实际上这不是必需的。

❸ 在退出之前，确保彻底关闭了消费者。

4.10 反序列化器

在之前的章节里提到过，生产者需要用序列化器把对象转换成字节数组再发送给 Kafka。类似地，消费者需要用反序列化器把从 Kafka 接收到的字节数组转换成 Java 对象。在前面的例子里，我们假设每个消息的键值对都是字符串，所以我们使用了默认的 String Deserializer。

在上一章讲解 Kafka 生产者的时候，我们已经介绍了如何序列化自定义对象类型，以及如何使用 Avro 和 AvroSerializer 根据定义好的 schema 生成 Avro 对象。现在来看看如何为对象自定义反序列化器，以及如何使用 Avro 和 Avro 反序列化器。

很显然，生成消息使用的序列化器与读取消息使用的反序列化器应该是一一对应的。使用 **IntSerializer** 序列化，然后使用

StringDeserializer 进行反序列化，会出现不可预测的结果。对于开发者来说，必须知道写入主题的消息使用的是哪一种序列化器，并确保每个主题里只包含能够被反序列化器解析的数据。使用 Avro 和 schema 注册表进行序列化和反序列化的优势在于：**AvroSerializer** 可以保证写入主题的数据与主题的 schema 是兼容的，也就是说，可以使用相应的反序列化器和 schema 来反序列化数据。另外，在生产者或消费者里出现的任何一个与兼容性有关的错误都会被捕捉到，它们都带有消息描述，也就是说，在出现序列化错误时，就没必要再去调试字节数组了。

尽管不建议使用自定义的反序列化器，我们仍然会简单地演示如何自定义反序列化器，然后再举例演示如何使用 Avro 来反序列化消息的键和值。

01. 自定义反序列化器

我们以在第 3 章使用过的自定义对象为例，为它写一个反序列化器。

```
public class Customer {
        private int customerID;
        private String customerName;

        public Customer(int ID, String name) {
                this.customerID = ID;
                this.customerName = name;
        }

  public int getID() {
    return customerID;
  }

  public String getName() {
    return customerName;
  }
}
```

自定义反序列化器看起来是这样的：

```
import org.apache.kafka.common.errors.SerializationException;

import java.nio.ByteBuffer;
import java.util.Map;

public class CustomerDeserializer implements
```

```
Deserializer<Customer> { ❶

      @Override
public void configure(Map configs, boolean isKey) {
 // 不需要做任何配置
}

@Override
public Customer deserialize(String topic, byte[] data) {

  int id;
  int nameSize;
  String name;

  try {
    if (data == null)
      return null;
    if (data.length < 8)
      throw new SerializationException("Size of data received by
        IntegerDeserializer is shorter than expected");

    ByteBuffer buffer = ByteBuffer.wrap(data);
    id = buffer.getInt();
    nameSize = buffer.getInt();

    byte[] nameBytes = new byte[nameSize];
    buffer.get(nameBytes);
    name = new String(nameBytes, "UTF-8");

    return new Customer(id, name); ❷

  } catch (Exception e) {
   throw new SerializationException("Error when serializing
     Customer
     to byte[] " + e);
  }
}

@Override
public void close() {
        // 不需要关闭任何东西
}
}
```

❶ 消费者也需要使用 Customer 类，这个类和序列化器在生产者和消费者应用程序里要相互匹配。在一个大型的企业里，会有很多消费者和生产者共享这些数据，这对于企业来说算是一个挑战。

❷ 我们把序列化器的逻辑反过来，从字节数组里获取

customer ID 和 name，再用它们构建需要的对象。

使用反序列化器的消费者代码看起来是这样的：

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.deserializer",
   "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.deserializer",
   "org.apache.kafka.common.serialization.CustomerDeserializer");

KafkaConsumer<String, Customer> consumer =
  new KafkaConsumer<>(props);

consumer.subscribe(Collections.singletonList("customerCountries"))

while (true) {
    ConsumerRecords<String, Customer> records =
      consumer.poll(100);
    for (ConsumerRecord<String, Customer> record : records)
    {
    System.out.println("current customer Id: " +
    record.value().getID() + " and
     current customer name: " + record.value().getName());
    }
}
```

再强调一次，我们并不建议使用自定义序列化器和自定义反序列化器。它们把生产者和消费者紧紧地耦合在一起，并且很脆弱，容易出错。我们建议使用标准的消息格式，比如 JSON、Thrift、Protobuf 或 Avro。接下来，我们会看到如何在消费者里使用 Avro 反序列化器。可以回到第 3 章查看有关 Avro 的项目背景、schema 和 schema 的兼容能力。

02. 在消费者里进行 **Avro** 反序列化

我们在 Avro 里使用第 3 章出现过的 Customer 类，为了读取这些对象，需要实现一个类似这样的消费者应用程序：

```
Properties props = new Properties();
props.put("bootstrap.servers", "broker1:9092,broker2:9092");
props.put("group.id", "CountryCounter");
props.put("key.serializer",
   "org.apache.kafka.common.serialization.StringDeserializer");
props.put("value.serializer",
   "io.confluent.kafka.serializers.KafkaAvroDeserializer"); ❶
```

```
props.put("schema.registry.url", schemaUrl); ❷
String topic = "customerContacts"

KafkaConsumer consumer = new
   KafkaConsumer(createConsumerConfig(brokers,groupId, url));
consumer.subscribe(Collections.singletonList(topic));

System.out.println("Reading topic:" + topic);

while (true) {
    ConsumerRecords<String, Customer> records =
      consumer.poll(1000); ❸

    for (ConsumerRecord<String, Customer> record: records) {
        System.out.println("Current customer name is: " +
           record.value().getName()); ❹
    }
    consumer.commitSync();
}
```

❶ 使用 KafkaAvroDeserializer 来反序列化 Avro 消息。

❷ `schema.registry.url` 是一个新的参数，它指向 schema 的存放位置。消费者可以使用由生产者注册的 schema 来反序列化消息。

❸ 将生成的类 `Customer` 作为值的类型。

❹ `record.value()` 返回的是一个 `Customer` 实例，接下来就可以使用它了。

4.11　独立消费者——为什么以及怎样使用没有群组的消费者

到目前为止，我们讨论了消费者群组，分区被自动分配给群组里的消费者，在群组里新增或移除消费者时自动触发再均衡。通常情况下，这些行为刚好是你所需要的，不过有时候你需要一些更简单的东西。比如，你可能只需要一个消费者从一个主题的所有分区或者某个特定的分区读取数据。这个时候就不需要消费者群组和再均衡了，只需要把主题或者分区分配给消费者，然后开始读取消息并提交偏移量。

如果是这样的话，就不需要订阅主题，取而代之的是为自己分配分区。一个消费者可以订阅主题（并加入消费者群组），或者为自己分

配分区，但不能同时做这两件事情。

下面的例子演示了一个消费者是如何为自己分配分区并从分区里读取消息的：

```
List<PartitionInfo> partitionInfos = null;
partitionInfos = consumer.partitionsFor("topic"); ❶

if (partitionInfos != null) {
    for (PartitionInfo partition : partitionInfos)
        partitions.add(new TopicPartition(partition.topic(),
          partition.partition()));
    consumer.assign(partitions); ❷

    while (true) {
        ConsumerRecords<String, String> records =
          consumer.poll(1000);

        for (ConsumerRecord<String, String> record: records) {
            System.out.println("topic = %s, partition = %s, offset = %d,
              customer = %s, country = %s\n",
              record.topic(), record.partition(), record.offset(),
              record.key(), record.value());
        }
        consumer.commitSync();
    }
}
```

❶ 向集群请求主题可用的分区。如果只打算读取特定分区，可以跳过这一步。

❷ 知道需要哪些分区之后，调用 `assign()` 方法。

除了不会发生再均衡，也不需要手动查找分区，其他的看起来一切正常。不过要记住，如果主题增加了新的分区，消费者并不会收到通知。所以，要么周期性地调用 `consumer.partitionsFor()` 方法来检查是否有新分区加入，要么在添加新分区后重启应用程序。

4.12　旧版的消费者API

我们在这一章讨论的 Java `KafkaConsumer` 客户端是 `org.apache.kafka.clients` 包的一部分。在本书写到这一章的时候，Kafka 还有两个旧版本的 Scala 消费者客户端，它们是 `kafka.consumer` 包的一部分，属于 Kafka 核心模块。它们分别被

叫作 `SimpleConsumer`（简单消费者，实际上也不是那么简单，它们是对 Kafka API 的轻度包装，可以用于从特定的分区和偏移量开始读取消息）和高级消费者。高级消费者指的就是 `ZookeeperConsumerConnector`，它有点像现在的消费者，有消费者群组，有分区再均衡，不过它使用 Zookeeper 来管理消费者群组，并不具备提交偏移量和再均衡的可操控性。

因为现在的消费者同时支持以上两种行为，并且为开发人员提供了更高的可靠性和可操控性，所以我们不打算讨论旧版 API。如果你想使用它们，那么请三思，如果确定要使用，可以从 Kafka 文档中了解更多的信息。

4.13　总结

我们在本章开头解释了 Kafka 消费者群组概念，消费者群组支持多个消费者从主题上读取消息。在介绍完概念之后，我们又给出了一个消费订阅主题并持续读取消息的例子。然后介绍了一些重要的消费者配置参数以及它们对消费者行为的影响。我们用一大部分内容解释偏移量以及消费者是如何管理偏移量的。了解消费者提交偏移量的方式有助更好地使用消费者客户端，所以我们介绍了几种不同的偏移量提交方式。然后又探讨了消费者 API 的其他话题，比如如何处理再均衡以及如何关闭消费者。

最后，我们介绍了反序列化器，消费者客户端使用反序列化器将保存在 Kafka 里的字节转换成应用程序可以处理的 Java 对象。我们主要介绍了 Avro 反序列化器，虽然有很多可用的反序列化器，但在 Kafka 里，Avro 是最为常用的一个。

现在，我们已经知道如何生成和读取 Kafka 消息，下一章将介绍 Kafka 内部的实现细节。

第 5 章　深入 Kafka

如果只是为了开发 Kafka 应用程序，或者只是在生产环境使用 Kafka，那么了解 Kafka 的内部工作原理不是必需的。不过，了解 Kafka 的内部工作原理有助于理解 Kafka 的行为，也有助于诊断问题。本章并不会涵盖 Kafka 的每一个设计和实现细节，而是集中讨论

以下 3 个有意思的话题：

- Kafka 如何进行复制；
- Kafka 如何处理来自生产者和消费者的请求；
- Kafka 的存储细节，比如文件格式和索引。

在对 Kafka 进行调优时，深入理解这些问题是很有必要的。了解了内部机制，可以更有目的性地进行深入的调优，而不只是停留在表面，隔靴搔痒。

5.1　集群成员关系

Kafka 使用 Zookeeper 来维护集群成员的信息。每个 broker 都有一个唯一标识符，这个标识符可以在配置文件里指定，也可以自动生成。在 broker 启动的时候，它通过创建临时节点把自己的 ID 注册到 Zookeeper。Kafka 组件订阅 Zookeeper 的 /brokers/ids 路径（broker 在 Zookeeper 上的注册路径），当有 broker 加入集群或退出集群时，这些组件就可以获得通知。

如果你要启动另一个具有相同 ID 的 broker，会得到一个错误——新 broker 会试着进行注册，但不会成功，因为 Zookeeper 里已经有一个具有相同 ID 的 broker。

在 broker 停机、出现网络分区或长时间垃圾回收停顿时，broker 会从 Zookeeper 上断开连接，此时 broker 在启动时创建的临时节点会自动从 Zookeeper 上移除。监听 broker 列表的 Kafka 组件会被告知该 broker 已移除。

在关闭 broker 时，它对应的节点也会消失，不过它的 ID 会继续存在于其他数据结构中。例如，主题的副本列表（下面会介绍）里就可能包含这些 ID。在完全关闭一个 broker 之后，如果使用相同的 ID 启动另一个全新的 broker，它会立即加入集群，并拥有与旧 broker 相同的分区和主题。

5.2　控制器

控制器其实就是一个 broker，只不过它除了具有一般 broker 的功能之外，还负责分区首领的选举（我们将在 5.3 节讨论分区首领选举）。集群里第一个启动的 broker 通过在 Zookeeper 里创建一个临时节点 `/controller` 让自己成为控制器。其他 broker 在启动时也

会尝试创建这个节点，不过它们会收到一个“节点已存在”的异常，然后“意识”到控制器节点已存在，也就是说集群里已经有一个控制器了。其他 broker 在控制器节点上创建 Zookeeper `watch` 对象，这样它们就可以收到这个节点的变更通知。这种方式可以确保集群里一次只有一个控制器存在。

如果控制器被关闭或者与 Zookeeper 断开连接，Zookeeper 上的临时节点就会消失。集群里的其他 broker 通过 `watch` 对象得到控制器节点消失的通知，它们会尝试让自己成为新的控制器。第一个在 Zookeeper 里成功创建控制器节点的 broker 就会成为新的控制器，其他节点会收到“节点已存在”的异常，然后在新的控制器节点上再次创建 `watch` 对象。每个新选出的控制器通过 Zookeeper 的条件递增操作获得一个全新的、数值更大的 `controller epoch`。其他 broker 在知道当前 `controller epoch` 后，如果收到由控制器发出的包含较旧 epoch 的消息，就会忽略它们。

当控制器发现一个 broker 已经离开集群（通过观察相关的 Zookeeper 路径），它就知道，那些失去首领的分区需要一个新首领（这些分区的首领刚好是在这个 broker 上）。控制器遍历这些分区，并确定谁应该成为新首领（简单来说就是分区副本列表里的下一个副本），然后向所有包含新首领或现有跟随者的 broker 发送请求。该请求消息包含了谁是新首领以及谁是分区跟随者的信息。随后，新首领开始处理来自生产者和消费者的请求，而跟随者开始从新首领那里复制消息。

当控制器发现一个 broker 加入集群时，它会使用 broker ID 来检查新加入的 broker 是否包含现有分区的副本。如果有，控制器就把变更通知发送给新加入的 broker 和其他 broker，新 broker 上的副本开始从首领那里复制消息。

简而言之，Kafka 使用 Zookeeper 的临时节点来选举控制器，并在节点加入集群或退出集群时通知控制器。控制器负责在节点加入或离开集群时进行分区首领选举。控制器使用 epoch 来避免“脑裂”。“脑裂”是指两个节点同时认为自己是当前的控制器。

5.3　复制

复制功能是 Kafka 架构的核心。在 Kafka 的文档里，Kafka 把自己描述成“一个分布式的、可分区的、可复制的提交日志服务”。复制之所以这么关键，是因为它可以在个别节点失效时仍能保证 Kafka 的可用

性和持久性。

Kafka 使用主题来组织数据，每个主题被分为若干个分区，每个分区有多个副本。那些副本被保存在 broker 上，每个 broker 可以保存成百上千个属于不同主题和分区的副本。

副本有以下两种类型。

首领副本

每个分区都有一个首领副本。为了保证一致性，所有生产者请求和消费者请求都会经过这个副本。

跟随者副本

首领以外的副本都是跟随者副本。跟随者副本不处理来自客户端的请求，它们唯一的任务就是从首领那里复制消息，保持与首领一致的状态。如果首领发生崩溃，其中的一个跟随者会被提升为新首领。

首领的另一个任务是搞清楚哪个跟随者的状态与自己是一致的。跟随者为了保持与首领的状态一致，在有新消息到达时尝试从首领那里复制消息，不过有各种原因会导致同步失败。例如，网络拥塞导致复制变慢，broker 发生崩溃导致复制滞后，直到重启 broker 后复制才会继续。

为了与首领保持同步，跟随者向首领发送获取数据的请求，这种请求与消费者为了读取消息而发送的请求是一样的。首领将响应消息发给跟随者。请求消息里包含了跟随者想要获取消息的偏移量，而且这些偏移量总是有序的。

一个跟随者副本先请求消息 1，接着请求消息 2，然后请求消息 3，在收到这 3 个请求的响应之前，它是不会发送第 4 个请求消息的。如果跟随者发送了请求消息 4，那么首领就知道它已经收到了前面 3 个请求的响应。通过查看每个跟随者请求的最新偏移量，首领就会知道每个跟随者复制的进度。如果跟随者在 10s 内没有请求任何消息，或者虽然在请求消息，但在 10s 内没有请求最新的数据，那么它就会被认为是不同步的。如果一个副本无法与首领保持一致，在首领发生失效时，它就不可能成为新首领——毕竟它没有包含全部的消息。

相反，持续请求得到的最新消息副本被称为同步的副本。在首领发生失效时，只有同步副本才有可能被选为新首领。

跟随者的正常不活跃时间或在成为不同步副本之前的时间是通过 `replica.lag.time.max.ms` 参数来配置的。这个时间间隔直接影响着首领选举期间的客户端行为和数据保留机制。我们将在第 6 章讨论可靠性保证，到时候会深入讨论这个问题。

除了当前首领之外，每个分区都有一个首选首领——创建主题时选定的首领就是分区的首选首领。之所以把它叫作首选首领，是因为在创建分区时，需要在 broker 之间均衡首领（后面会介绍在 broker 间分布副本和首领的算法）。因此，我们希望首选首领在成为真正的首领时，broker 间的负载最终会得到均衡。默认情况下，Kafka 的 `auto.leader.rebalance.enable` 被设为 `true`，它会检查首选首领是不是当前首领，如果不是，并且该副本是同步的，那么就会触发首领选举，让首选首领成为当前首领。

> 找到首选首领
>
> 从分区的副本清单里可以很容易找到首选首领（可以使用 kafka.topics.sh 工具查看副本和分区的详细信息，我们将在第 10 章介绍管理工具）。清单里的第一个副本一般就是首选首领。不管当前首领是哪一个副本，都不会改变这个事实，即使使用副本分配工具将副本重新分配给其他 broker。要记住，如果你手动进行副本分配，第一个指定的副本就是首选首领，所以要确保首选首领被传播到其他 broker 上，避免让包含了首领的 broker 负载过重，而其他 broker 却无法为它们分担负载。

5.4 处理请求

broker 的大部分工作是处理客户端、分区副本和控制器发送给分区首领的请求。Kafka 提供了一个二进制协议（基于 TCP），指定了请求消息的格式以及 broker 如何对请求作出响应——包括成功处理请求或在处理请求过程中遇到错误。客户端发起连接并发送请求， broker 处理请求并作出响应。broker 按照请求到达的顺序来处理它们——这种顺序保证让 Kafka 具有了消息队列的特性，同时保证保存的消息也是有序的。

所有的请求消息都包含一个标准消息头：

- Request type（也就是 API key）

- Request version（broker 可以处理不同版本的客户端请求，并根据客户端版本作出不同的响应）
- Correlation ID——一个具有唯一性的数字，用于标识请求消息，同时也会出现在响应消息和错误日志里（用于诊断问题）
- Client ID——用于标识发送请求的客户端

我们不打算在这里描述该协议，因为在 Kafka 文档里已经有很详细的说明。不过，了解 broker 如何处理请求还是有必要的——后面在我们讨论 Kafka 监控和各种配置选项时，你就会了解到那些与队列和线程有关的度量指标和配置参数。

broker 会在它所监听的每一个端口上运行一个 `Acceptor` 线程，这个线程会创建一个连接，并把它交给 `Processor` 线程去处理。`Processor` 线程（也被叫作“网络线程”）的数量是可配置的。网络线程负责从客户端获取请求消息，把它们放进请求队列，然后从响应队列获取响应消息，把它们发送给客户端。图 5-1 为 Kafka 处理请求的内部流程。

请求消息被放到请求队列后，**IO** 线程会负责处理它们。下面是几种最常见的请求类型。

生产请求

　　生产者发送的请求，它包含客户端要写入 broker 的消息。

获取请求

　　在消费者和跟随者副本需要从 broker 读取消息时发送的请求。

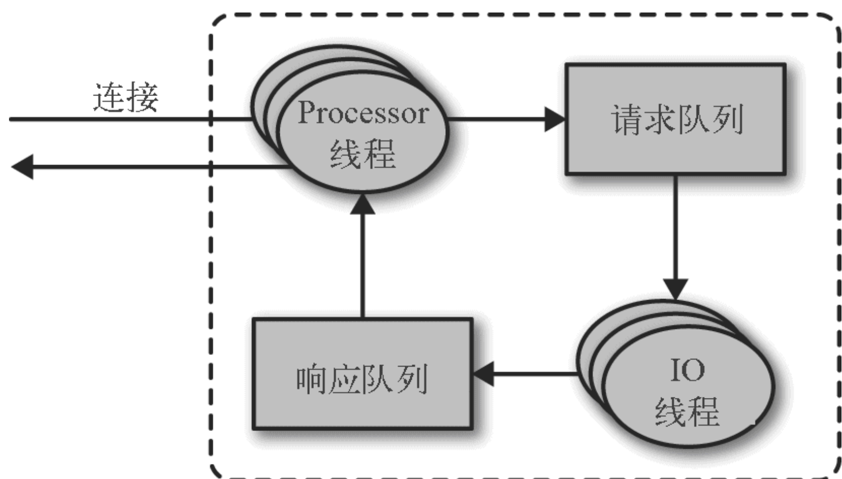


图 5-1：**Kafka** 处理请求的内部流程

生产请求和获取请求都必须发送给分区的首领副本。如果 broker 收到一个针对特定分区的请求，而该分区的首领在另一个 broker 上，那么发送请求的客户端会收到一个“非分区首领”的错误响应。当针对特定分区的获取请求被发送到一个不含有该分区首领的 broker 上，也会出现同样的错误。Kafka 客户端要自己负责把生产请求和获取请求发送到正确的 broker 上。

那么客户端怎么知道该往哪里发送请求呢？客户端使用了另一种请求类型，也就是元数据请求。这种请求包含了客户端感兴趣的主题列表。服务器端的响应消息里指明了这些主题所包含的分区、每个分区都有哪些副本，以及哪个副本是首领。元数据请求可以发送给任意一个 broker，因为所有 broker 都缓存了这些信息。

一般情况下，客户端会把这些信息缓存起来，并直接往目标 broker 上发送生产请求和获取请求。它们需要时不时地通过发送元数据请求来刷新这些信息（刷新的时间间隔通过 `metadata.max.age.ms` 参数来配置），从而知道元数据是否发生了变更——比如，在新 broker 加入集群时，部分副本会被移动到新的 broker 上（如图 5-2 所示）。另外，如果客户端收到“非首领”错误，它会在尝试重发请求之前先刷新元数据，因为这个错误说明了客户端正在使用过期的元数据信息，之前的请求被发到了错误的 broker 上。

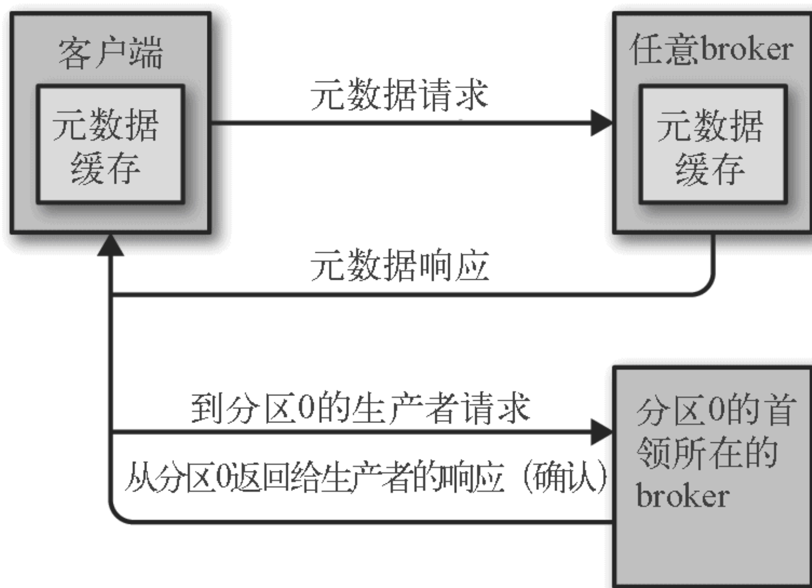


图 **5-2**：客户端路由请求

5.4.1　生产请求

我们在第 3 章讨论如何配置生产者的时候，提到过 `acks` 这个配置参数——该参数指定了需要多少个 broker 确认才可以认为一个消息写入是成功的。不同的配置对“写入成功”的界定是不一样的，如果 `acks=1`，那么只要首领收到消息就认为写入成功；如果 `acks=all`，那么需要所有同步副本收到消息才算写入成功；如果 `acks=0`，那么生产者在把消息发出去之后，完全不需要等待 broker 的响应。

包含首领副本的 broker 在收到生产请求时，会对请求做一些验证。

- 发送数据的用户是否有主题写入权限？
- 请求里包含的 `acks` 值是否有效（只允许出现 `0`、`1` 或 `all`）？
- 如果 `acks=all`，是否有足够多的同步副本保证消息已经被安全写入？（我们可以对 broker 进行配置，如果同步副本的数量不足，broker 可以拒绝处理新消息。在第 6 章介绍 Kafka 持久性和可靠性保证时，我们会讨论更多这方面的细节。）

之后，消息被写入本地磁盘。在 Linux 系统上，消息会被写到文件系统缓存里，并不保证它们何时会被刷新到磁盘上。Kafka 不会一直等待数据被写到磁盘上——它依赖复制功能来保证消息的持久性。

在消息被写入分区的首领之后，broker 开始检查 `acks` 配置参数——如果 `acks` 被设为 `0` 或 `1`，那么 broker 立即返回响应；如果 `acks` 被设为 `all`，那么请求会被保存在一个叫作炼狱的缓冲区里，直到首领发现所有跟随者副本都复制了消息，响应才会被返回给客户端。

5.4.2　获取请求

broker 处理获取请求的方式与处理生产请求的方式很相似。客户端发送请求，向 broker 请求主题分区里具有特定偏移量的消息，好像在说："请把主题 Test 分区 0 偏移量从 53 开始的消息以及主题 Test 分区 3 偏移量从 64 开始的消息发给我。"客户端还可以指定 broker 最多可以从一个分区里返回多少数据。这个限制是非常重要的，因为客户端需要为 broker 返回的数据分配足够的内存。如果没有这个限制，broker 返回的大量数据有可能耗尽客户端的内存。

我们之前讨论过，请求需要先到达指定的分区首领上，然后客户端通过查询元数据来确保请求的路由是正确的。首领在收到请求时，它会先检查请求是否有效——比如，指定的偏移量在分区上是否存在？如果客户端请求的是已经被删除的数据，或者请求的偏移量不存在，那么 broker 将返回一个错误。

如果请求的偏移量存在，broker 将按照客户端指定的数量上限从分区里读取消息，再把消息返回给客户端。Kafka 使用零复制技术向客户端发送消息——也就是说，Kafka 直接把消息从文件（或者更确切地说是 Linux 文件系统缓存）里发送到网络通道，而不需要经过任何中间缓冲区。这是 Kafka 与其他大部分数据库系统不一样的地方，其他数据库在将数据发送给客户端之前会先把它们保存在本地缓存里。这项技术避免了字节复制，也不需要管理内存缓冲区，从而获得更好的性能。

客户端除了可以设置 broker 返回数据的上限，也可以设置下限。例如，如果把下限设置为 10KB，就好像是在告诉 broker："等到有 10KB 数据的时候再把它们发送给我。"在主题消息流量不是很大的情况下，这样可以减少 CPU 和网络开销。客户端发送一个请求，broker 等到有足够的数据时才把它们返回给客户端，然后客户端再发出请求，而不是让客户端每隔几毫秒就发送一次请求，每次只能得到

很少的数据甚至没有数据。（如图 5-3 所示。）对比这两种情况，它们最终读取的数据总量是一样的，但前者的来回传送次数更少，因此开销也更小。

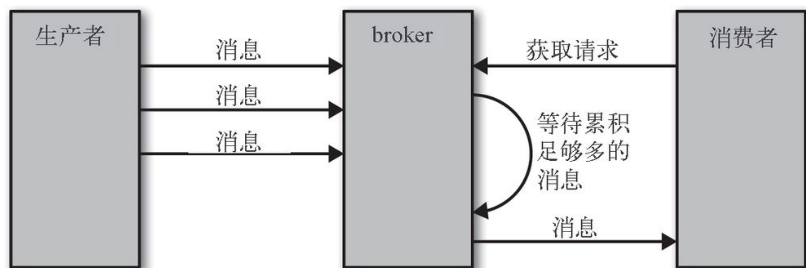


图 **5-3**：**broker** 延迟作出响应以便累积足够的数据

当然，我们不会让客户端一直等待 broker 累积数据。在等待了一段时间之后，就可以把可用的数据拿回处理，而不是一直等待下去。所以，客户端可以定义一个超时时间，告诉 broker："如果你无法在 *X* 毫秒内累积满足要求的数据量，那么就把当前这些数据返回给我。"

有意思的是，并不是所有保存在分区首领上的数据都可以被客户端读取。大部分客户端只能读取已经被写入所有同步副本的消息（跟随者副本也不行，尽管它们也是消费者——否则复制功能就无法工作）。分区首领知道每个消息会被复制到哪个副本上，在消息还没有被写入所有同步副本之前，是不会发送给消费者的——尝试获取这些消息的请求会得到空的响应而不是错误。

因为还没有被足够多副本复制的消息被认为是"不安全"的——如果首领发生崩溃，另一个副本成为新首领，那么这些消息就丢失了。如果我们允许消费者读取这些消息，可能就会破坏一致性。试想，一个消费者读取并处理了这样的一个消息，而另一个消费者发现这个消息其实并不存在。所以，我们会等到所有同步副本复制了这些消息，才允许消费者读取它们（如图 5-4 所示）。这也意味着，如果 broker 间的消息复制因为某些原因变慢，那么消息到达消费者的时间也会随之变长（因为我们会先等待消息复制完毕）。延迟时间可以通过参数 `replica.lag.time.max.ms` 来配置，它指定了副本在复制消息时可被允许的最大延迟时间。

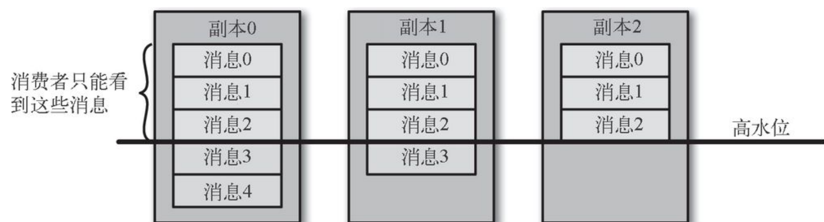


图 **5-4**：消费者只能看到已经复制到 **ISR** 的消息

5.4.3　其他请求

到此为止，我们讨论了 Kafka 最为常见的几种请求类型：元数据请求、生产请求和获取请求。重要的是，我们讨论的是客户端在网络上使用的通用二进制协议。Kafka 内置了由开源社区贡献者实现和维护的 Java 客户端，同时也有用其他语言实现的客户端，如 C、Python、Go 语言等。Kafka 网站上有它们的完整清单，这些客户端就是使用这个二进制协议与 broker 通信的。

另外，broker 之间也使用同样的通信协议。它们之间的请求发生在 Kafka 内部，客户端不应该使用这些请求。例如，当一个新首领被选举出来，控制器会发送 `LeaderAndIsr` 请求给新首领（这样它就可以开始接收来自客户端的请求）和跟随者（这样它们就知道要开始跟随新首领）。

在我们写这本书的时候，Kafka 协议可以处理 20 种不同类型的请求，而且会有更多的类型加入进来。协议在持续演化——随着客户端功能的不断增加，我们需要改进协议来满足需求。例如，之前的 Kafka 消费者使用 Zookeeper 来跟踪偏移量，在消费者启动的时候，它通过检查保存在 Zookeeper 上的偏移量就可以知道从哪里开始处理消息。因为各种原因，我们决定不再使用 Zookeeper 来保存偏移量，而是把偏移量保存在特定的 Kafka 主题上。为了达到这个目的，我们不得不往协议里增加几种请求类型：`OffsetCommitRequest`、`OffsetFetchRequest`和 `ListOffsetsRequest`。现在，在应用程序调用 `commitOffset()` 方法时，客户端不再把偏移量写入 Zookeeper，而是往 Kafka 发送 `OffsetCommitRequest` 请求。

主题的创建仍然需要通过命令行工具来完成，命令行工具会直接更新 Zookeeper 里的主题列表，broker 监听这些主题列表，在有新主题加入时，它们会收到通知。我们正在改进 Kafka，增加了 `CreateTopicRequest` 请求类型，这样客户端（包括那些不支持

Zookeeper 客户端的编程语言）就可以直接向 broker 请求创建新主题了。

除了往协议里增加新的请求类型外，我们也会通过修改已有的请求类型来给它们增加新功能。例如，从 Kafka 0.9.0 到 Kafka 0.10.0，我们希望能够让客户端知道谁是当前的控制器，于是把控制器信息添加到元数据响应消息里。我们还在元数据请求消息和响应消息里添加了一个新的 version 字段。现在，0.9.0 版本的客户端发送的元数据请求里 version 为 0（0.9.0 版本客户端的 version 不会是 1）。不管是 0.9.0 版本的 broker，还是 0.10.0 版本的 broker，它们都知道应该返回 version 为 0 的响应，也就是不包含控制器信息的响应。0.9.0 版本的客户端不需要控制器的信息，而且也没必要知道如何去解析它。0.10.0 版本的客户端会发送 version 为 1 的元数据请求，0.10.0 版本的 broker 会返回 version 为 1 的响应，里面包含了控制器的信息。如果 0.10.0 版本的客户端发送 version 为 1 的请求给 0.9.0 版本的 broker，这个版本的 broker 不知道该如何处理这个请求，就会返回一个错误。这就是为什么我们建议在升级客户端之前先升级 broker，因为新的 broker 知道如何处理旧的请求，反过来则不然。

我们在 0.10.0 版本的 Kafka 里加入了 `ApiVersionRequest`——客户端可以询问 broker 支持哪些版本的请求，然后使用正确的版本与 broker 通信。如果能够正确使用这个新功能，客户端就可以与旧版本的 broker 通信，只要 broker 支持这个版本的协议。

5.5　物理存储

Kafka 的基本存储单元是分区。分区无法在多个 broker 间进行再细分，也无法在同一个 broker 的多个磁盘上进行再细分。所以，分区的大小受到单个挂载点可用空间的限制（一个挂载点由单个磁盘或多个磁盘组成，如果配置了 JBOD，就是单个磁盘，如果配置了 RAID，就是多个磁盘。请参考第 2 章）。

在配置 Kafka 的时候，管理员指定了一个用于存储分区的目录清单——也就是 `log.dirs` 参数的值（不要把它与存放错误日志的目录混淆了，日志目录是配置在 log4j.properties 文件里的）。该参数一般会包含每个挂载点的目录。

接下来我们会介绍 Kafka 是如何使用这些目录来存储数据的。首先，我们要知道数据是如何被分配到集群的 broker 上以及 broker 的目录里的。然后，我们还要知道 broker 是如何管理这些文件的，特别是

如何进行数据保留的。随后，我们会深入探讨文件和索引格式。最后，我们会讨论日志压缩及其工作原理。日志压缩是 Kafka 的一个高级特性，因为有了这个特性，Kafka 可以用来长时间地保存数据。

5.5.1　分区分配

在创建主题时，Kafka 首先会决定如何在 broker 间分配分区。假设你有 6 个 broker，打算创建一个包含 10 个分区的主题，并且复制系数为 3。那么 Kafka 就会有 30 个分区副本，它们可以被分配给 6 个 broker。在进行分区分配时，我们要达到如下的目标。

- 在 broker 间平均地分布分区副本。对于我们的例子来说，就是要保证每个 broker 可以分到 5 个副本。
- 确保每个分区的每个副本分布在不同的 broker 上。假设分区 0 的首领副本在 broker 2 上，那么可以把跟随者副本放在 broker 3 和 broker 4 上，但不能放在 broker 2 上，也不能两个都放在 broker 3 上。
- 如果为 broker 指定了机架信息，那么尽可能把每个分区的副本分配到不同机架的 broker 上。这样做是为了保证一个机架的不可用不会导致整体的分区不可用。

为了实现这个目标，我们先随机选择一个 broker（假设是 4），然后使用轮询的方式给每个 broker 分配分区来确定首领分区的位置。于是，首领分区 0 会在 broker 4 上，首领分区 1 会在 broker 5 上，首领分区 2 会在 broker 0 上（只有 6 个 broker），并以此类推。然后，我们从分区首领开始，依次分配跟随者副本。如果分区 0 的首领在 broker 4 上，那么它的第一个跟随者副本会在 broker 5 上，第二个跟随者副本会在 broker 0 上。分区 1 的首领在 broker 5 上，那么它的第一个跟随者副本在 broker 0 上，第二个跟随者副本在 broker 1 上。

如果配置了机架信息，那么就不是按照数字顺序来选择 broker 了，而是按照交替机架的方式来选择 broker。假设 broker 0、broker 1 和 broker 2 放置在同一个机架上，broker 3、broker 4 和 broker 5 分别放置在其他不同的机架上。我们不是按照从 0 到 5 的顺序来选择 broker，而是按照 0，3，1，4，2，5 的顺序来选择，这样每个相邻的 broker 都在不同的机架上（如图 5-5 所示）。于是，如果分区 0 的首领在 broker 4 上，那么第一个跟随者副本会在 broker 2 上，这两个 broker 在不同的机架上。如果第一个机架下线，还有其他副本仍然活跃着，所以分区仍然可用。这对所有副本来说都是一样的，因

此在机架下线时仍然能够保证可用性。

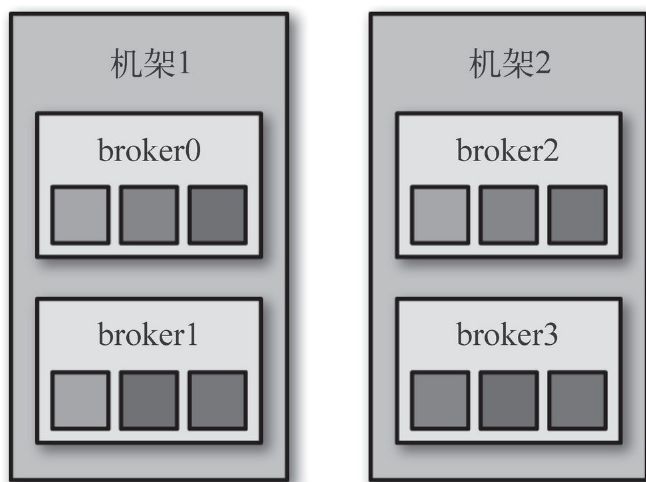


图 **5-5**：分配给不同机架 **broker** 的分区和副本

为分区和副本选好合适的 broker 之后，接下来要决定这些分区应该使用哪个目录。我们单独为每个分区分配目录，规则很简单：计算每个目录里的分区数量，新的分区总是被添加到数量最小的那个目录里。也就是说，如果添加了一个新磁盘，所有新的分区都会被创建到这个磁盘上。因为在完成分配工作之前，新磁盘的分区数量总是最少的。

 注意磁盘空间

要注意，在为 broker 分配分区时并没有考虑可用空间和工作负载问题，但在将分区分配到磁盘上时会考虑分区数量，不过不考虑分区大小。也就是说，如果有些 broker 的磁盘空间比其他 broker 要大（有可能是因为集群同时使用了旧服务器和新服务器），有些分区异常大，或者同一个 broker 上有大小不同的磁盘，那么在分配分区时要格外小心。在后面的章节中，我们会讨论 Kafka 管理员该如何解决这种 broker 负载不均衡的问题。

5.5.2　文件管理

保留数据是 Kafka 的一个基本特性，Kafka 不会一直保留数据，也不

会等到所有消费者都读取了消息之后才删除消息。相反，Kafka 管理员为每个主题配置了数据保留期限，规定数据被删除之前可以保留多长时间，或者清理数据之前可以保留的数据量大小。

因为在一个大文件里查找和删除消息是很费时的，也很容易出错，所以我们把分区分成若干个片段。默认情况下，每个片段包含 1GB 或一周的数据，以较小的那个为准。在 broker 往分区写入数据时，如果达到片段上限，就关闭当前文件，并打开一个新文件。

当前正在写入数据的片段叫作活跃片段。活动片段永远不会被删除，所以如果你要保留数据 1 天，但片段里包含了 5 天的数据，那么这些数据会被保留 5 天，因为在片段被关闭之前这些数据无法被删除。如果你要保留数据一周，而且每天使用一个新片段，那么你就会看到，每天在使用一个新片段的同时会删除一个最老的片段——所以大部分时间该分区会有 7 个片段存在。

我们在第 2 章讲过，broker 会为分区里的每个片段打开一个文件句柄，哪怕片段是不活跃的。这样会导致打开过多的文件句柄，所以操作系统必须根据实际情况做一些调优。

5.5.3　文件格式

我们把 Kafka 的消息和偏移量保存在文件里。保存在磁盘上的数据格式与从生产者发送过来或者发送给消费者的消息格式是一样的。因为使用了相同的消息格式进行磁盘存储和网络传输，Kafka 可以使用零复制技术给消费者发送消息，同时避免了对生产者已经压缩过的消息进行解压和再压缩。

除了键、值和偏移量外，消息里还包含了消息大小、校验和、消息格式版本号、压缩算法（Snappy、GZip 或 LZ4）和时间戳（在 0.10.0 版本里引入的）。时间戳可以是生产者发送消息的时间，也可以是消息到达 broker 的时间，这个是可配置的。

如果生产者发送的是压缩过的消息，那么同一个批次的消息会被压缩在一起，被当作“包装消息”进行发送（如图 5-6 所示）。于是，broker 就会收到一个这样的消息，然后再把它发送给消费者。消费者在解压这个消息之后，会看到整个批次的消息，它们都有自己的时间戳和偏移量。

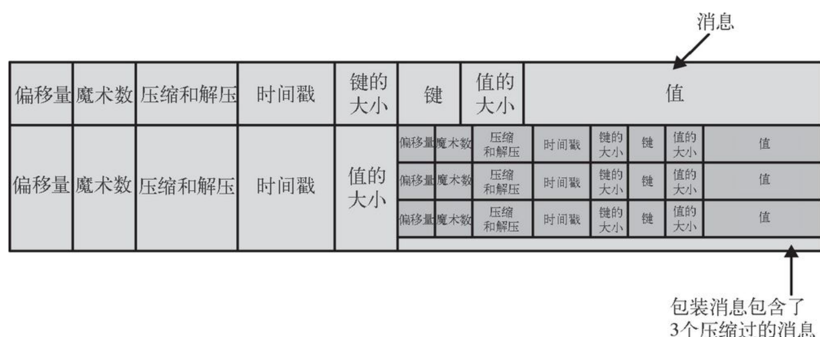


图 5-6：普通消息和包装消息

也就是说，如果在生产者端使用了压缩功能（极力推荐），那么发送的批次越大，就意味着在网络传输和磁盘存储方面会获得越好的压缩性能，同时意味着如果修改了消费者使用的消息格式（例如，在消息里增加了时间戳），那么网络传输和磁盘存储的格式也要随之修改，而且 broker 要知道如何处理包含了两种消息格式的文件。

Kafka 附带了一个叫 DumpLogSegment 的工具，可以用它查看片段的内容。它可以显示每个消息的偏移量、校验和、魔术数字节、消息大小和压缩算法。运行该工具的方法如下：

```
bin/kafka-run-class.sh kafka.tools.DumpLogSegments
```

如果使用了 `--deep-iteration` 参数，可以显示被压缩到包装消息里的消息。

5.5.4　索引

消费者可以从 Kafka 的任意可用偏移量位置开始读取消息。假设消费者要读取从偏移量 100 开始的 1MB 消息，那么 broker 必须立即定位到偏移量 100（可能是在分区的任意一个片段里），然后开始从这个位置读取消息。为了帮助 broker 更快地定位到指定的偏移量，Kafka 为每个分区维护了一个索引。索引把偏移量映射到片段文件和偏移量在文件里的位置。

索引也被分成片段，所以在删除消息时，也可以删除相应的索引。Kafka 不维护索引的校验和。如果索引出现损坏，Kafka 会通过重新读取消息并录制偏移量和位置来重新生成索引。如果有必要，管理员可以删除索引，这样做是绝对安全的，Kafka 会自动重新生成这些索

引。

5.5.5 清理

一般情况下，Kafka 会根据设置的时间保留数据，把超过时效的旧数据删除掉。不过，试想一下这样的场景，如果你使用 Kafka 保存客户的收货地址，那么保存客户的最新地址比保存客户上周甚至去年的地址要有意义得多，这样你就不用担心会用错旧地址，而且短时间内客户也不会修改新地址。另外一个场景，一个应用程序使用 Kafka 保存它的状态，每次状态发生变化，它就把状态写入 Kafka。在应用程序从崩溃中恢复时，它从 Kafka 读取消息来恢复最近的状态。在这种情况下，应用程序只关心它在崩溃前的那个状态，而不关心运行过程中的那些状态。

Kafka 通过改变主题的保留策略来满足这些使用场景。早于保留时间的旧事件会被删除，为每个键保留最新的值，从而达到清理的效果。很显然，只有当应用程序生成的事件里包含了键值对时，为这些主题设置 `compact` 策略才有意义。如果主题包含 `null` 键，清理就会失败。

5.5.6 清理的工作原理

每个日志片段可以分为以下两个部分。

干净的部分

　　这些消息之前被清理过，每个键只有一个对应的值，这个值是上一次清理时保留下来的。

污浊的部分

　　这些消息是在上一次清理之后写入的。两个部分的日志片段示意如图 5-7 所示。

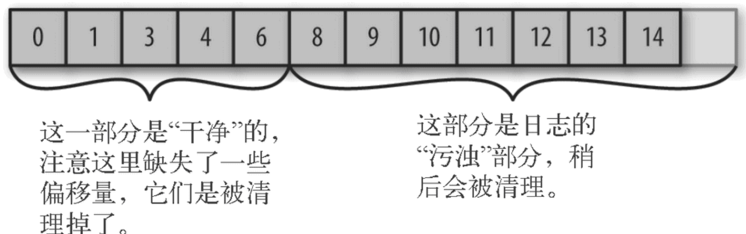


图 **5-7**：包含干净和污浊两个部分的分区

如果在 Kafka 启动时启用了清理功能（通过配置 `log.cleaner.enabled` 参数），每个 broker 会启动一个清理管理器线程和多个清理线程，它们负责执行清理任务。这些线程会选择污浊率（污浊消息占分区总大小的比例）较高的分区进行清理。

为了清理分区，清理线程会读取分区的污浊部分，并在内存里创建一个 map。map 里的每个元素包含了消息键的散列值和消息的偏移量，键的散列值是 16B，加上偏移量总共是 24B。如果要清理一个 1GB 的日志片段，并假设每个消息大小为 1KB，那么这个片段就包含一百万个消息，而我们只需要用 24MB 的 map 就可以清理这个片段。（如果有重复的键，可以重用散列项，从而使用更少的内存。）这是非常高效的！

管理员在配置 Kafka 时可以对 map 使用的内存大小进行配置。每个线程都有自己的 map，而这个参数指的是所有线程可使用的内存总大小。如果你为 map 分配了 1GB 内存，并使用了 5 个清理线程，那么每个线程可以使用 200MB 内存来创建自己的 map。Kafka 并不要求分区的整个污浊部分来适应这个 map 的大小，但要求至少有一个完整的片段必须符合。如果不符合，那么 Kafka 就会报错，管理员要么分配更多的内存，要么减少清理线程数量。如果只有少部分片段可以完全符合，Kafka 将从最旧的片段开始清理，等待下一次清理剩余的部分。

清理线程在创建好偏移量 map 后，开始从干净的片段处读取消息，从最旧的消息开始，把它们的内容与 map 里的内容进行比对。它会检查消息的键是否存在于 map 中，如果不存在，那么说明消息的值是最新的，就把消息复制到替换片段上。如果键已存在，消息会被忽略，因为在分区的后部已经有一个具有相同键的消息存在。在复制完所有的消息之后，我们就将替换片段与原始片段进行交换，然后开始清理下一个片段。完成整个清理过程之后，每个键对应一个不同的消息——这些消息的值都是最新的。清理前后的分区片段如图 5-8 所示。

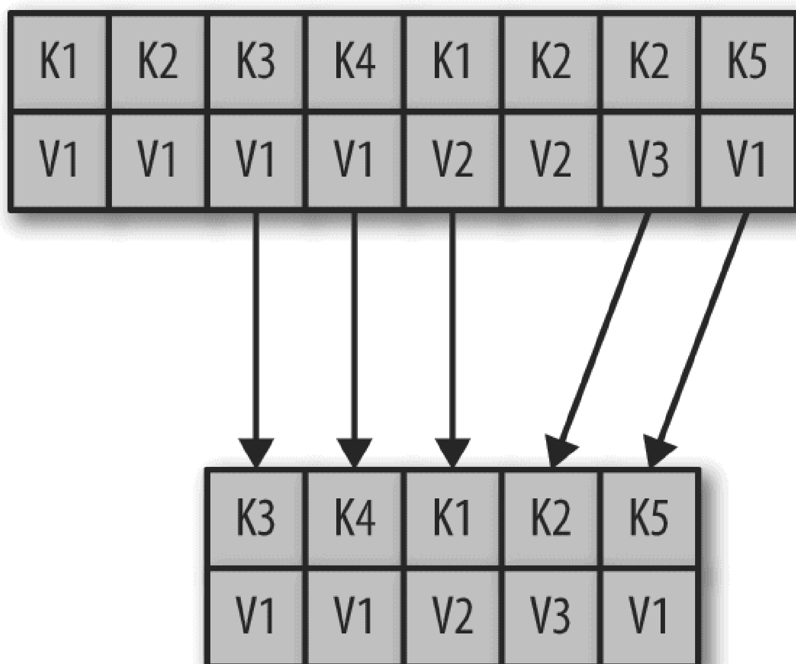


图 **5-8**：清理前后的分区片段

5.5.7　被删除的事件

如果只为每个键保留最近的一个消息，那么当需要删除某个特定键所对应的所有消息时，我们该怎么办？这种情况是有可能发生的，比如一个用户不再使用我们的服务，那么完全可以把与这个用户相关的所有信息从系统中删除。

为了彻底把一个键从系统里删除，应用程序必须发送一个包含该键且值为 `null` 的消息。清理线程发现该消息时，会先进行常规的清理，只保留值为 `null` 的消息。该消息（被称为墓碑消息）会被保留一段时间，时间长短是可配置的。在这期间，消费者可以看到这个墓碑消息，并且发现它的值已经被删除。于是，如果消费者往数据库里复制 Kafka 的数据，当它看到这个墓碑消息时，就知道应该要把相关的用户信息从数据库里删除。在这个时间段过后，清理线程会移除这个墓碑消息，这个键也将从 Kafka 分区里消失。重要的是，要留给消费者足够多的时间，让他看到墓碑消息，因为如果消费者离线几个小时并错过了墓碑消息，就看不到这个键，也就不知道它已经从 Kafka 里删

除，从而也就不会去删除数据库里的相关数据了。

5.5.8　何时会清理主题

就像 `delete` 策略不会删除当前活跃的片段一样，`compact` 策略也不会对当前片段进行清理。只有旧片段里的消息才会被清理。

在 0.10.0 和更早的版本里，Kafka 会在包含脏记录的主题数量达到 50% 时进行清理。这样做的目的是避免太过频繁的清理（因为清理会影响主题的读写性能），同时也避免存在太多脏记录（因为它们会占用磁盘空间）。浪费 50% 的磁盘空间给主题存放脏记录，然后进行一次清理，这是个合理的折中，管理员也可以对它进行调整。

我们计划在未来的版本中加入宽限期，在宽限期内，我们保证消息不会被清理。对于想看到主题的每个消息的应用程序来说，它们就有了足够的时间，即使时间有点滞后。

5.6　总结

我们无法在这一章里涵盖所有的内容，但希望大家能够对我们在这个项目上所做的设计和优化有所了解，同时本章也为大家解释了在使用 Kafka 时可能碰到的一些晦涩难懂的现象和参数配置问题。

如果大家真的对 Kafka 内部原理感兴趣，唯一的途径是阅读它的源代码。Kafka 开发者邮件组（dev@kafka.apache.org）是一个非常友好的社区，在那里会有人回答有关 Kafka 工作原理的问题。或许你在阅读源代码时还能够修复一些缺陷——开源社区的大门总是向贡献者敞开。

第 6 章　可靠的数据传递

对于系统来说，可靠的数据传递不能成为马后炮。与性能一样，在系统的设计之初就应该考虑可靠性问题，而不能在事后才来考虑。而且，可靠性是系统的一个属性，而不是一个独立的组件，所以在讨论 Kafka 的可靠性保证时，还是要从系统的整体出发。说到可靠性，那些与 Kafka 集成的系统与 Kafka 本身一样重要。正因为可靠性是系统层面的概念，所以它不只是某个个体的事情。Kafka 管理员、Linux

系统管理员、网络和存储管理员以及应用程序开发者，所有人必须协同作战，才能构建一个可靠的系统。

Kafka 在数据传递可靠性方面具备很大的灵活性。我们知道，Kafka 可以被用在很多场景里，从跟踪用户点击动作到处理信用卡支付操作。有些场景要求很高的可靠性，而有些则更看重速度和简便性。Kafka 被设计成高度可配置的，而且它的客户端 API 可以满足不同程度的可靠性需求。

不过，灵活性有时候也很容易让人掉入陷阱。有时候，你的系统看起来是可靠的，但实际上有可能不是。本章先讨论各种各样的可靠性及其在 Kafka 场景中的含义。然后介绍 Kafka 的复制功能，以及它是如何提高系统可靠性的。随后探讨如何配置 Kafka 的 broker 和主题来满足不同的使用场景需求，也会涉及生产者和消费者以及如何在各种可靠性场景里使用它们。最后介绍如何验证系统的可靠性，因为系统的可靠性涉及方方面面——一些前提条件必须先得到满足。

6.1　可靠性保证

在讨论可靠性时，我们一般会使用保证这个词，它是指确保系统在各种不同的环境下能够发生一致的行为。

ACID 大概是大家最熟悉的一个例子，它是关系型数据库普遍支持的标准可靠性保证。

ACID 指的是原子性、一致性、隔离性和持久性。如果一个供应商说他们的数据库遵循 ACID 规范，其实就是在说他们的数据库支持与事务相关的行为。

有了这些保证，我们才能相信关系型数据库的事务特性可以确保应用程序的安全。我们知道系统承诺可以做到些什么，也知道在不同条件下它们会发生怎样的行为。我们了解这些保证机制，并基于这些保证机制开发安全的应用程序。

所以，了解系统的保证机制对于构建可靠的应用程序来说至关重要，这也是能够在不同条件下解释系统行为的前提。那么 Kafka 可以在哪些方面作出保证呢？

- Kafka 可以保证分区消息的顺序。如果使用同一个生产者往同一个分区写入消息，而且消息 B 在消息 A 之后写入，那么 Kafka 可以保证消息 B 的偏移量比消息 A 的偏移量大，而且消

费者会先读取消息 A 再读取消息 B。
- 只有当消息被写入分区的所有同步副本时（但不一定要写入磁盘），它才被认为是“已提交”的。生产者可以选择接收不同类型的确认，比如在消息被完全提交时的确认，或者在消息被写入首领副本时的确认，或者在消息被发送到网络时的确认。
- 只要还有一个副本是活跃的，那么已经提交的消息就不会丢失。
- 消费者只能读取已经提交的消息。

这些基本的保证机制可以用来构建可靠的系统，但仅仅依赖它们是无法保证系统完全可靠的。构建一个可靠的系统需要作出一些权衡，Kafka 管理员和开发者可以在配置参数上作出权衡，从而得到他们想要达到的可靠性。这种权衡一般是指消息存储的可靠性和一致性的重要程度与可用性、高吞吐量、低延迟和硬件成本的重要程度之间的权衡。下面将介绍 Kafka 的复制机制，并探讨 Kafka 是如何实现可靠性的，最后介绍一些重要的配置参数。

6.2　复制

Kafka 的复制机制和分区的多副本架构是 Kafka 可靠性保证的核心。把消息写入多个副本可以使 Kafka 在发生崩溃时仍能保证消息的持久性。

我们已经在第 5 章深入解释了 Kafka 的复制机制，现在重新回顾一下主要内容。

Kafka 的主题被分为多个分区，分区是基本的数据块。分区存储在单个磁盘上，Kafka 可以保证分区里的事件是有序的，分区可以在线（可用），也可以离线（不可用）。每个分区可以有多个副本，其中一个副本是首领。所有的事件都直接发送给首领副本，或者直接从首领副本读取事件。其他副本只需要与首领保持同步，并及时复制最新的事件。当首领副本不可用时，其中一个同步副本将成为新首领。

分区首领是同步副本，而对于跟随者副本来说，它需要满足以下条件才能被认为是同步的。

- 与 Zookeeper 之间有一个活跃的会话，也就是说，它在过去的 6s（可配置）内向 Zookeeper 发送过心跳。
- 在过去的 10s 内（可配置）从首领那里获取过消息。
- 在过去的 10s 内从首领那里获取过最新的消息。光从首领那里

获取消息是不够的，它还必须是几乎零延迟的。

如果跟随者副本不能满足以上任何一点，比如与 Zookeeper 断开连接，或者不再获取新消息，或者获取消息滞后了 10s 以上，那么它就被认为是不同步的。一个不同步的副本通过与 Zookeeper 重新建立连接，并从首领那里获取最新消息，可以重新变成同步的。这个过程在网络出现临时问题并很快得到修复的情况下会很快完成，但如果 broker 发生崩溃就需要较长的时间。

非同步副本

如果一个或多个副本在同步和非同步状态之间快速切换，说明集群内部出现了问题，通常是 Java 不恰当的垃圾回收配置导致的。不恰当的垃圾回收配置会造成几秒钟的停顿，从而让 broker 与 Zookeeper 之间断开连接，最后变成不同步的，进而发生状态切换。

一个滞后的同步副本会导致生产者和消费者变慢，因为在消息被认为已提交之前，客户端会等待所有同步副本接收消息。而如果一个副本不再同步了，我们就不再关心它是否已经收到消息。虽然非同步副本同样滞后，但它并不会对性能产生任何影响。但是，更少的同步副本意味着更低的有效复制系数，在发生宕机时丢失数据的风险更大。

我们将在下一节讲解在实际项目中这将意味着什么。

6.3 broker配置

broker 有 3 个配置参数会影响 Kafka 消息存储的可靠性。与其他配置参数一样，它们可以应用在 broker 级别，用于控制所有主题的行为，也可以应用在主题级别，用于控制个别主题的行为。

在主题级别控制可靠性，意味着 Kafka 集群可以同时拥有可靠的主题和非可靠的主题。例如，在银行里，管理员可能把整个集群设置为可靠的，但把其中的一个主题设置为非可靠的，用于保存来自客户的投诉，因为这些消息是允许丢失的。

让我们来逐个介绍这些配置参数，看看它们如何影响消息存储的可靠性，以及 Kafka 在哪些方面作出了权衡。

6.3.1 复制系数

主题级别的配置参数是 `replication.factor`，而在 broker 级别则可以通过 `default.replication.factor` 来配置自动创建的主题。

在这本书里，我们假设主题的复制系数都是 3，也就是说每个分区总共会被 3 个不同的 broker 复制 3 次。这样的假设是合理的，因为 Kafka 的默认复制系数就是 3——不过用户可以修改它。即使是在主题创建之后，也可以通过新增或移除副本来改变复制系数。

如果复制系数为 *N*，那么在 *N*-1 个 broker 失效的情况下，仍然能够从主题读取数据或向主题写入数据。所以，更高的复制系数会带来更高的可用性、可靠性和更少的故障。另一方面，复制系数 *N* 需要至少 *N* 个 broker，而且会有 *N* 个数据副本，也就是说它们会占用 *N* 倍的磁盘空间。我们一般会在可用性和存储硬件之间作出权衡。

那么该如何确定一个主题需要几个副本呢？这要看主题的重要程度，以及你愿意付出多少成本来换取可用性。有时候这与你的偏执程度也有点关系。

如果因 broker 重启导致的主题不可用是可接受的（这在集群里是很正常的行为），那么把复制系数设为 1 就可以了。在作出这个权衡的时候，要确保这样不会对你的组织和用户造

成影响，因为你在节省了硬件成本的同时也降低了可用性。复制系数为 2 意味着可以容忍 1 个 broker 发生失效，看起来已经足够了。不过要记住，有时候 1 个 broker 发生失效会导致集群不稳定（通常是旧版的 Kafka），迫使你重启另一个 broker——集群控制器。也就是说，如果将复制系数设为 2，就有可能因为重启等问题导致集群不可用。所以这是一个两难的选择。

基于以上几点原因，我们建议在要求可用性的场景里把复制系数设为 3。在大多数情况下，这已经足够安全了——不过我们也见过有些银行使用 5 个副本，以防不测。

副本的分布也很重要。默认情况下，Kafka 会确保分区的每个副本被放在不同的 broker 上。不过，有时候这样仍然不够安全。如果这些 broker 处于同一个机架上，一旦机架的交换机发生故障，分区就会不可用，这时候把复制系数设为多少都不管用。为了避免机架级别的故障，我们建议把 broker 分布在多个不同的机架上，并使用 `broker.rack` 参数来为每个 broker 配置所在机架的名字。如果配置了机架名字，Kafka 会保证分区的副本被分布在多个机架上，从而

获得更高的可用性。我们已经在第 5 章介绍了如何在 broker 和机架上分布副本，如果你对此感兴趣，可以参考第 5 章的内容。

6.3.2　不完全的首领选举

`unclean.leader.election` 只能在 broker 级别（实际上是在集群范围内）进行配置，它的默认值是 `true`。

我们之前提到过，当分区首领不可用时，一个同步副本会被选为新首领。如果在选举过程中没有丢失数据，也就是说提交的数据同时存在于所有的同步副本上，那么这个选举就是“完全”的。

但如果在首领不可用时其他副本都是不同步的，我们该怎么办呢？这种情况会在以下两种场景里出现。

- 分区有 3 个副本，其中的两个跟随者副本不可用（比如有两个 broker 发生崩溃）。这个时候，如果生产者继续往首领写入数据，所有消息都会得到确认并被提交（因为此时首领是唯一的同步副本）。现在我们假设首领也不可用了（又一个 broker 发生崩溃），这个时候，如果之前的一个跟随者重新启动，它就成为了分区的唯一不同步副本。
- 分区有 3 个副本，因为网络问题导致两个跟随者副本复制消息滞后，所以尽管它们还在复制消息，但已经不同步了。首领作为唯一的同步副本继续接收消息。这个时候，如果首领变为不可用，另外两个副本就再也无法变成同步的了。

对于这两种场景，我们要作出一个两难的选择。

- 如果不同步的副本不能被提升为新首领，那么分区在旧首领（最后一个同步副本）恢复之前是不可用的。有时候这种状态会持续数小时（比如更换内存芯片）。
- 如果不同步的副本可以被提升为新首领，那么在这个副本变为不同步之后写入旧首领的消息会全部丢失，导致数据不一致。为什么会这样呢？假设在副本 0 和副本 1 不可用时，偏移量 100~200 的消息被写入副本 2（首领）。现在副本 2 变为不可用的，而副本 0 变为可用的。副本 0 只包含偏移量 0~100 的消息，不包含偏移量 100~200 的消息。如果我们允许副本 0 成为新首领，生产者就可以继续写入数据，消费者可以继续读取数据。于是，新首领就有了偏移量 100~200 的新消息。这样，部分消费者会读取到偏移量 100~200 的旧消息，部分消费者会读取到偏移量 100~200 的新消息，还有部分消费者读

取的是二者的混合。这样会导致非常不好的结果，比如生成不准确的报表。另外，副本 2 可能会重新变为可用，并成为新首领的跟随者。这个时候，它会把比当前首领旧的消息全部删除，而这些消息对于所有消费者来说都是不可用的。

简而言之，如果我们允许不同步的副本成为首领，那么就要承担丢失数据和出现数据不一致的风险。如果不允许它们成为首领，那么就要接受较低的可用性，因为我们必须等待原先的首领恢复到可用状态。

如果把 `unclean.leader.election.enable` 设为 `true`，就是允许不同步的副本成为首领（也就是“不完全的选举”），那么我们将面临丢失消息的风险。如果把这个参数设为 `false`，就要等待原先的首领重新上线，从而降低了可用性。我们经常看到一些对数据质量和数据一致性要求较高的系统会禁用这种不完全的首领选举（把这个参数设为 `false`）。银行系统是这方面最好的例子，大部分银行系统宁愿选择在几分钟甚至几个小时内不处理信用卡支付事务，也不会冒险处理错误的消息。不过在对可用性要求较高的系统里，比如实时点击流分析系统，一般会启用不完全的首领选举。

6.3.3　最少同步副本

在主题级别和 broker 级别上，这个参数都叫 `min.insync.replicas`。

我们知道，尽管为一个主题配置了 3 个副本，还是会出现只有一个同步副本的情况。如果这个同步副本变为不可用，我们必须在可用性和一致性之间作出选择——这是一个两难的选择。根据 Kafka 对可靠性保证的定义，消息只有在被写入到所有同步副本之后才被认为是已提交的。但如果这里的“所有副本”只包含一个同步副本，那么在这个副本变为不可用时，数据就会丢失。

如果要确保已提交的数据被写入不止一个副本，就需要把最少同步副本数量设置为大一点的值。对于一个包含 3 个副本的主题，如果 `min.insync.replicas` 被设为 2，那么至少要存在两个同步副本才能向分区写入数据。

如果 3 个副本都是同步的，或者其中一个副本变为不可用，都不会有什么问题。不过，如果有两个副本变为不可用，那么 broker 就会停止接受生产者的请求。尝试发送数据的生产者会收到 `NotEnoughReplicasException` 异常。消费者仍然可以继续读取已有的数据。实际上，如果使用这样的配置，那么当只剩下一个同步

副本时，它就变成只读了，这是为了避免在发生不完全选举时数据的写入和读取出现非预期的行为。为了从只读状态中恢复，必须让两个不可用分区中的一个重新变为可用的（比如重启 broker），并等待它变为同步的。

6.4 在可靠的系统里使用生产者

即使我们尽可能把 broker 配置得很可靠，但如果没有对生产者进行可靠性方面的配置，整个系统仍然有可能出现突发性的数据丢失。

请看以下两个例子。

- 为 broker 配置了 3 个副本，并且禁用了不完全首领选举，这样应该可以保证万无一失。我们把生产者发送消息的 `acks` 设为 `1`（只要首领接收到消息就可以认为消息写入成功）。生产者发送一个消息给首领，首领成功写入，但跟随者副本还没有接收到这个消息。首领向生产者发送了一个响应，告诉它“消息写入成功”，然后它崩溃了，而此时消息还没有被其他副本复制过去。另外两个副本此时仍然被认为是同步的（毕竟判定一个副本不同步需要一小段时间），而且其中的一个副本成了新的首领。因为消息还没有被写入这个副本，所以就丢失了，但发送消息的客户端却认为消息已成功写入。因为消费者看不到丢失的消息，所以此时的系统仍然是一致的（因为副本没有收到这个消息，所以消息不算已提交），但从生产者角度来看，它丢失了一个消息。
- 为 broker 配置了 3 个副本，并且禁用了不完全首领选举。我们接受了之前的教训，把生产者的 `acks` 设为 `all`。假设现在往 Kafka 发送消息，分区的首领刚好崩溃，新的首领正在选举当中，Kafka 会向生产者返回“首领不可用”的响应。在这个时候，如果生产者没能正确处理这个错误，也没有重试发送消息直到发送成功，那么消息也有可能丢失。这算不上是 broker 的可靠性问题，因为 broker 并没有收到这个消息。这也不是一致性问题，因为消费者并没有读到这个消息。问题在于如果生产者没能正确处理这些错误，弄丢消息的是它们自己。

那么，我们该如何避免这些悲剧性的后果呢？从上面两个例子可以看出，每个使用 Kafka

的开发人员都要注意两件事情。

- 根据可靠性需求配置恰当的 `acks` 值。
- 在参数配置和代码里正确处理错误。

第 3 章已经深入讨论了生产者的几种模式，现在回顾几个要点。

6.4.1　发送确认

生产者可以选择以下 3 种不同的确认模式。

- `acks=0` 意味着如果生产者能够通过网络把消息发送出去，那么就认为消息已成功写入 Kafka。在这种情况下还是有可能发生错误，比如发送的对象无法被序列化或者网卡发生故障，但如果是分区离线或整个集群长时间不可用，那就不会收到任何错误。即使是在发生完全首领选举的情况下，这种模式仍然会丢失消息，因为在新首领选举过程中它并不知道首领已经不可用了。在 `acks=0` 模式下的运行速度是非常快的（这就是为什么很多基准测试都是基于这个模式），你可以得到惊人的吞吐量和带宽利用率，不过如果选择了这种模式，一定会丢失一些消息。
- `acks=1` 意味着首领在收到消息并把它写入到分区数据文件（不一定同步到磁盘上）时会返回确认或错误响应。在这个模式下，如果发生正常的首领选举，生产者会在选举时收到一个 `LeaderNotAvailableException` 异常，如果生产者能恰当地处理这个错误（参考 6.4.2 节），它会重试发送消息，最终消息会安全到达新的首领那里。不过在这个模式下仍然有可能丢失数据，比如消息已经成功写入首领，但在消息被复制到跟随者副本之前首领发生崩溃。
- `acks=all` 意味着首领在返回确认或错误响应之前，会等待所有同步副本都收到消息。如果和 `min.insync.replicas` 参数结合起来，就可以决定在返回确认前至少有多少个副本能够收到消息。这是最保险的做法——生产者会一直重试直到消息被成功提交。不过这也是最慢的做法，生产者在继续发送其他消息之前需要等待所有副本都收到当前的消息。可以通过使用异步模式和更大的批次来加快速度，但这样做通常会降低吞吐量。

6.4.2　配置生产者的重试参数

生产者需要处理的错误包括两部分：一部分是生产者可以自动处理的错误，还有一部分是需要开发者手动处理的错误。

如果 broker 返回的错误可以通过重试来解决，那么生产者会自动处理这些错误。生产者向 broker 发送消息时，broker 可以返回一个成功响应码或者一个错误响应码。错误响应码可以分为两种，一种是在重试之后可以解决的，还有一种是无法通过重试解决的。例如，如果 broker 返回的是 `LEADER_NOT_AVAILABLE` 错误，生产者可以尝试重新发送消息。也许在这个时候一个新的首领被选举出来了，那么这次发送就会成功。也就是说，`LEADER_NOT_AVAILABLE` 是一个可重试错误。另一方面，如果 broker 返回的是 `INVALID_CONFIG` 错误，即使通过重试也无法改变配置选项，所以这样的重试是没有意义的。这种错误是不可重试错误。

一般情况下，如果你的目标是不丢失任何消息，那么最好让生产者在遇到可重试错误时能够保持重试。为什么要这样？因为像首领选举或网络连接这类问题都可以在几秒钟之内得到解决，如果让生产者保持重试，你就不需要额外去处理这些问题了。经常会有人问："为生产者配置多少重试次数比较好？"这个要看你在生产者放弃重试并抛出异常之后想做些什么。如果你想抓住异常并再多重试几次，那么就可以把重试次数设置得多一点，让生产者继续重试；如果你想直接丢弃消息，多次重试造成的延迟已经失去发送消息的意义；如果你想把消息保存到某个地方然后回过头来再继续处理，那就可以停止重试。Kafka 的跨数据中心复制工具（MirrorMaker，我们将在第 8 章介绍）默认会进行无限制的重试（例如 `retries=MAX_INT`）。作为一个具有高可靠性的复制工具，它决不会丢失消息。

要注意，重试发送一个已经失败的消息会带来一些风险，如果两个消息都写入成功，会导致消息重复。例如，生产者因为网络问题没有收到 broker 的确认，但实际上消息已经写入成功，生产者会认为网络出现了临时故障，就重试发送该消息（因为它不知道消息已经写入成功）。在这种情况下，broker 会收到两个相同的消息。重试和恰当的错误处理可以保证每个消息"至少被保存一次"，但当前的 Kafka 版本（0.10.0）无法保证每个消息"只被保存一次"。现实中的很多应用程序在消息里加入唯一标识符，用于检测重复消息，消费者在读取消息时可以对它们进行清理。还要一些应用程序可以做到消息的"幂等"，也就是说，即使出现了重复消息，也不会对处理结果的正确性造成负面影响。例如，消息"这个账号里有 110 美元"就是幂等的，因为即使多次发送这样的消息，产生的结果都是一样的。不过消息"往这个账号里增加 10 美元"就不是幂等的。

6.4.3　额外的错误处理

使用生产者内置的重试机制可以在不造成消息丢失的情况下轻松地处理大部分错误，不过对于开发人员来说，仍然需要处理其他类型的错误，包括：

- 不可重试的 broker 错误，例如消息大小错误、认证错误等；
- 在消息发送之前发生的错误，例如序列化错误；
- 在生产者达到重试次数上限时或者在消息占用的内存达到上限时发生的错误。

我们在第 3 章讨论了如何为同步发送消息和异步发送消息编写错误处理器。这些错误处理器的代码逻辑与具体的应用程序及其目标有关。丢弃“不合法的消息”？把错误记录下来？把这些消息保存在本地磁盘上？回调另一个应用程序？具体使用哪一种逻辑要根据具体的架构来决定。只要记住，如果错误处理只是为了重试发送消息，那么最好还是使用生产者内置的重试机制。

6.5　在可靠的系统里使用消费者

我们已经学习了如何在保证 Kafka 可靠性的前提下生产数据，现在来看看如何在同样的前提下读取数据。

在本章的开始部分可以看到，只有那些被提交到 Kafka 的数据，也就是那些已经被写入所有同步副本的数据，对消费者是可用的，这意味着消费者得到的消息已经具备了一致性。消费者唯一要做的是跟踪哪些消息是已经读取过的，哪些是还没有读取过的。这是在读取消息时不丢失消息的关键。

在从分区读取数据时，消费者会获取一批事件，检查这批事件里最大的偏移量，然后从这个偏移量开始读取另外一批事件。这样可以保证消费者总能以正确的顺序获取新数据，不会错过任何事件。

如果一个消费者退出，另一个消费者需要知道从什么地方开始继续处理，它需要知道前一个消费者在退出前处理的最后一个偏移量是多少。所谓的“另一个”消费者，也可能就是它自己重启之后重新回来工作。这也就是为什么消费者要“提交”它们的偏移量。它们把当前读取的偏移量保存起来，在退出之后，同一个群组里的其他消费者就可以接手它们的工作。如果消费者提交了偏移量却未能处理完消息，那么就有可能造成消息丢失，这也是消费者丢失消息的主要原因。在这种情况下，如果其他消费者接手了工作，那些没有被处理完的消息就会被忽略，永远得不到处理。这就是为什么我们非常重视偏移量提交的

时间点和提交的方式。

> 已提交消息与已提交偏移量
>
> 要注意，此处的已提交消息与之前讨论过的已提交消息是不一样的，它是指已经被写入所有同步副本并且对消费者可见的消息，而已提交偏移量是指消费者发送给 Kafka 的偏移量，用于确认它已经收到并处理好的消息位置。

我们在第 4 章已经详细介绍了消费者 API 的使用，还介绍了多种提交偏移量的方式。下面会介绍一些关键的注意事项，如果要了解消费者 API 的使用细节，请参考第 4 章。

6.5.1　消费者的可靠性配置

为了保证消费者行为的可靠性，需要注意以下 4 个非常重要的配置参数。

第 1 个是 `group.id`。这个参数在第 4 章已经详细解释过了，如果两个消费者具有相同的 `group.id`，并且订阅了同一个主题，那么每个消费者会分到主题分区的一个子集，也就是说它们只能读到所有消息的一个子集（不过群组会读取主题所有的消息）。如果你希望消费者可以看到主题的所有消息，那么需要为它们设置唯一的 `group.id`。

第 2 个是 `auto.offset.reset`。这个参数指定了在没有偏移量可提交时（比如消费者第 1 次启动时）或者请求的偏移量在 broker 上不存在时（第 4 章已经解释过这种场景），消费者会做些什么。这个参数有两种配置。一种是 `earliest`，如果选择了这种配置，消费者会从分区的开始位置读取数据，不管偏移量是否有效，这样会导致消费者读取大量的重复数据，但可以保证最少的数据丢失。一种是 `latest`，如果选择了这种配置，消费者会从分区的末尾开始读取数据，这样可以减少重复处理消息，但很有可能会错过一些消息。

第 3 个是 `enable.auto.commit`。这是一个非常重要的配置参数，你可以让消费者基于任务调度自动提交偏移量，也可以在代码里手动提交偏移量。自动提交的一个最大好处是，在实现消费者逻辑时可以少考虑一些问题。如果你在消费者轮询操作里处理所有的数据，那么自动提交可以保证只提交已经处理过的偏移量（如果忘了消费者轮询是什么，请回顾一下第 4 章的内容）。自动提交的主要缺点是，

无法控制重复处理消息（比如消费者在自动提交偏移量之前停止处理消息），而且如果把消息交给另外一个后台线程去处理，自动提交机制可能会在消息还没有处理完毕就提交偏移量。

第 4 个配置参数 `auto.commit.interval.ms` 与第 3 个参数有直接的联系。如果选择了自动提交偏移量，可以通过该参数配置提交的频度，默认值是每 5 秒钟提交一次。一般来说，频繁提交会增加额外的开销，但也会降低重复处理消息的概率。

6.5.2　显式提交偏移量

如果选择了自动提交偏移量，就不需要关心显式提交的问题。不过如果希望能够更多地控制偏移量提交的时间点，那么就要仔细想想该如何提交偏移量了——要么是为了减少重复处理消息，要么是因为把消息处理逻辑放在了轮询之外。

这里我们不再重复说明这个机制以及如何使用相关的 API，因为第 4 章里已经有很详细的介绍。相反，我们会着重说明几个在开发具有可靠性的消费者应用程序时需要注意的事项。我们先从简单的开始，再逐步深入。

01. 总是在处理完事件后再提交偏移量

    如果所有的处理都是在轮询里完成，并且不需要在轮询之间维护状态（比如为了实现聚合操作），那么可以使用自动提交，或者在轮询结束时进行手动提交。

02. 提交频度是性能和重复消息数量之间的权衡

    即使是在最简单的场景里，比如所有的处理都在轮询里完成，并且不需要在轮询之间维护状态，你仍然可以在一个循环里多次提交偏移量（甚至可以在每处理完一个事件之后），或者多个循环里只提交一次（与生产者的 `acks=all` 配置有点类似），这完全取决于你在性能和重复处理消息之间作出的权衡。

03. 确保对提交的偏移量心里有数

    在轮询过程中提交偏移量有一个不好的地方，就是提交的偏移量有可能是读取到的最新偏移量，而不是处理过的最新偏移量。要记住，在处理完消息后再提交偏移量是非常关键的——

否则会导致消费者错过消息。我们已经在第 4 章给出了示例。

04. 再均衡

在设计应用程序时要注意处理消费者的再均衡问题。我们在第 4 章举了几个例子，一般要在分区被撤销之前提交偏移量，并在分配到新分区时清理之前的状态。

05. 消费者可能需要重试

有时候，在进行轮询之后，有些消息不会被完全处理，你想稍后再来处理。例如，假设要把 Kafka 的数据写到数据库里，不过那个时候数据库不可用，于是你想稍后重试。要注意，你提交的是偏移量，而不是对消息的“确认”，这个与传统的发布和订阅消息系统不太一样。如果记录 #30 处理失败，但记录 #31 处理成功，那么你不应该提交 #31，否则会导致 #31 以内的偏移量都被提交，包括 #30 在内，而这可能不是你想看到的结果。不过可以采用以下两种模式来解决这个问题。

第一种模式，在遇到可重试错误时，提交最后一个处理成功的偏移量，然后把还没有处理好的消息保存到缓冲区里（这样下一个轮询就不会把它们覆盖掉），调用消费者的 `pause()` 方法来确保其他的轮询不会返回数据（不需要担心在重试时缓冲区溢出），在保持轮询的同时尝试重新处理（关于为什么不能停止轮询，请参考第 4 章）。如果重试成功，或者重试次数达到上限并决定放弃，那么把错误记录下来并丢弃消息，然后调用 `resume()` 方法让消费者继续从轮询里获取新数据。

第二种模式，在遇到可重试错误时，把错误写入一个独立的主题，然后继续。一个独立的消费者群组负责从该主题上读取错误消息，并进行重试，或者使用其中的一个消费者同时从该主题上读取错误消息并进行重试，不过在重试时需要暂停该主题。这种模式有点像其他消息系统里的 dead-letter-queue。

06. 消费者可能需要维护状态

有时候你希望在多个轮询之间维护状态，例如，你想计算消息的移动平均数，希望在首次轮询之后计算平均数，然后在后续的轮询中更新这个结果。如果进程重启，你不仅需要从上一个偏移量开始处理数据，还要恢复移动平均数。有一种办法是在提交偏移量的同时把最近计算的平均数写到一个“结果”主题

上。消费者线程在重新启动之后，它就可以拿到最近的平均数并接着计算。不过这并不能完全地解决问题，因为 Kafka 并没有提供事务支持。消费者有可能在写入平均数之后来不及提交偏移量就崩溃了，或者反过来也一样。这是一个很复杂的问题，你不应该尝试自己去解决这个问题，建议尝试一下 KafkaStreams 这个类库，它为聚合、连接、时间窗和其他复杂的分析提供了高级的 DSL API。

07. 长时间处理

有时候处理数据需要很长时间：你可能会从发生阻塞的外部系统获取信息，或者把数据写到外部系统，或者进行一个非常复杂的计算。要记住，暂停轮询的时间不能超过几秒钟。即使不想获取更多的数据，也要保持轮询，这样客户端才能往 broker 发送心跳。在这种情况下，一种常见的做法是使用一个线程池来处理数据，因为使用多个线程可以进行并行处理，从而加快处理速度。在把数据移交给线程池去处理之后，你就可以暂停消费者，然后保持轮询，但不获取新数据，直到工作线程处理完成。在工作线程处理完成之后，可以让消费者继续获取新数据。因为消费者一直保持轮询，心跳会正常发送，就不会发生再均衡。

08. 仅一次传递

有些应用程序不仅仅需要“至少一次”（at-least-once）语义（意味着没有数据丢失），还需要“仅一次”（exactly-once）语义。尽管 Kafka 现在还不能完全支持仅一次语义，消费者还是有一些办法可以保证 Kafka 里的每个消息只被写到外部系统一次（但不会处理向 Kafka 写入数据时可能出现的重复数据）。

实现仅一次处理最简单且最常用的办法是把结果写到一个支持唯一键的系统里，比如键值存储引擎、关系型数据库、ElasticSearch 或其他数据存储引擎。在这种情况下，要么消息本身包含一个唯一键（通常都是这样），要么使用主题、分区和偏移量的组合来创建唯一键——它们的组合可以唯一标识一个 Kafka 记录。如果你把消息和一个唯一键写入系统，然后碰巧又读到一个相同的消息，只要把原先的键值覆盖掉即可。数据存储引擎会覆盖已经存在的键值对，就像没有出现过重复数据一样。这个模式被叫作幂等性写入，它是一种很常见也很有用的模式。

如果写入消息的系统支持事务，那么就可以使用另一种方法。最简单的是使用关系型数据库，不过 HDFS 里有一些被重新定义过的原子操作也经常用来达到相同的目的。我们把消息和偏移量放在同一个事务里，这样它们就能保持同步。在消费者启动时，它会获取最近处理过的消息偏移量，然后调用 `seek()` 方法从该偏移量位置继续读取数据。我们在第 4 章已经介绍了一个相关的例子。

6.6 验证系统可靠性

你经过了所有的流程，从确认可靠性需求，到配置 broker，再到配置客户端，并小心谨慎地使用 API……现在可以把所有东西都放到生产环境里去运行，然后高枕无忧，自信不会丢失任何消息了，对吗？

你当然可以这么做，不过建议还是先对系统可靠性做一些验证。我们建议做 3 个层面的验证——配置验证、应用程序验证以及生产环境的应用程序监控。让我们来看看每一步都要做些什么以及该怎么做。

6.6.1 配置验证

从应用程序里可以很容易对 broker 和客户端配置进行验证，我们之所以建议这么做，有以下两方面的原因。

- 验证配置是否满足你的需求。
- 帮助你理解系统的行为，了解系统的真正行为是什么，了解你对 Kafka 基本准则的理解是否存在偏差，然后加以改进，同时了解这些准则是如何被应用到各种场景里的。这一章的内容偏重理论，所以要确保你能够理解这些理论是如何运用于实际当中的。

Kafka 提供了两个重要的工具用于验证配置：org.apache.kafka.tools 包里的 `Verifiable Producer` 和 `VerifiableConsumer` 这两个类。我们可以从命令行运行这两个类，或者把它们嵌入到自动化测试框架里。

其思想是，`VerifiableProducer` 生成一系列消息，这些消息包含从 1 到你指定的某个数字。你可以使用与生产者相同的方式来配置 `VerifiableProducer`，比如配置相同的 `acks`、重试次数和消息生成速度。在运行 `VerifiableProducer` 时，它会把每个消息是否成功发送到 broker 的结果打印出来。`VerifiableConsumer` 执

行的是另一个检查——它读取事件（由 `VerifiableProducer` 生成）并按顺序打印出这些事件。它也会打印出已提交的偏移量和再均衡的相关信息。

你可以考虑运行以下一些测试。

- 首领选举：如果我停掉首领会发生什么事情？生产者和消费者重新恢复正常状态需要多长时间？
- 控制器选举：重启控制器后系统需要多少时间来恢复状态？
- 依次重启：可以依次重启 broker 而不丢失任何数据吗？
- 不完全首领选举测试：如果依次停止所有副本（确保每个副本都变为不同步的），然后启动一个不同步的 broker 会发生什么？要怎样恢复正常？这样做是可接受的吗？

然后你从中选择一个场景，启动 `VerifiableProducer` 和 `VerifiableConsumer` 并开始测试这个场景，例如，停掉正在接收消息的分区首领。如果期望在一个短暂的暂停之后状态恢复正常并且没有任何数据丢失，那么只要确保生产者生成的数据个数与消费者读取的数据个数是匹配的就可以了。

Kafka 的代码库里包含了大量测试用例。它们大部分都遵循相同的准则——使用 `Verifiable Producer` 和 `VerifiableConsumer` 来确保迭代的版本能够正常工作。

6.6.2　应用程序验证

在确定 broker 和客户端的配置可以满足你的需求之后，接下来要验证应用程序是否能够保证达到你的期望。应用程序的验证包括检查自定义的错误处理代码、偏移量提交的方式、再均衡监听器以及其他使用了 Kafka 客户端的地方。

因为应用程序是你自己的，关于如何测试应用程序的逻辑，我们无法提供更多的指导，但愿你的开发流程里已经包含了集成测试。不管如何验证你的应用程序，我们都建议基于如下的故障条件做一些测试：

- 客户端从服务器断开连接（系统管理员可以帮忙模拟网络故障）；
- 首领选举；
- 依次重启 broker；
- 依次重启消费者；
- 依次重启生产者。

你对每一个测试场景都会有期望的行为，也就是在开发应用程序时所期望看到的行为，然后运行测试看看真实的结果是否符合预期。例如，在测试“依次重启消费者”这一场景时，你期望看到“在短暂的再均衡之后出现的重复消息个数不超过 1000 个”。测试结果会告诉我们应用程序提交偏移量的方式和处理再均衡的方式是否与预期的一样。

6.6.3 在生产环境监控可靠性

测试应用程序是很重要的，不过它无法代替生产环境的持续监控，这些监控是为了确保数据按照期望的方式流动。我们将会在第 9 章详细介绍如何监控 Kafka 集群，不过除了监控集群的健康状况之外，监控客户端和数据流也是很重要的。

首先，Kafka 的 Java 客户端包含了 JMX 度量指标，这些指标可以用于监控客户端的状态和事件。对于生产者来说，最重要的两个可靠性指标是消息的 error-rate 和 retry-rate（聚合过的）。如果这两个指标上升，说明系统出现了问题。除此以外，还要监控生产者日志——发送消息的错误日志被设为 WARN 级别，可以在“Got error produce response with correlation id 5689 on topic-partition [topic-1,3], retrying (two attempts left). Error: ... ”中找到它们。如果你看到消息剩余的重试次数为 0，说明生产者已经没有多余的重试机会。就像我们在 6.4 节所讨论的那样，你也许可以增加重试次数，或者把造成这个错误的问题先解决掉。

对于消费者来说，最重要的指标是 consumer-lag，该指标表明了消费者的处理速度与最近提交到分区里的偏移量之间还有多少差距。理想情况下，该指标总是为 0，消费者总能读到最新的消息。不过在实际当中，因为 `poll()` 方法会返回很多消息，消费者在获取更多数据之前需要花一些时间来处理它们，所以该指标会有些波动。关键是要确保消费者最终会赶上去，而不是越落越远。因为该指标会正常波动，所以在告警系统里配置该指标有一定难度。Burrow 是 LinkedIn 公司开发的一个 consumer-lag 检测工具，它可以让这件事情变得容易一些。

监控数据流是为了确保所有生成的数据会被及时地读取（你的需求决定了“及时”的具体含义）。为了确保数据能够被及时读取，你需要知道数据是什么时候生成的。0.10.0 版本的 Kafka 在消息里增加了时间戳，表明了消息的生成时间。如果你使用的是更早版本的客户端，我们建议自己在消息里加入时间戳、应用程序的名字和机器名，这样有助于将来诊断问题。

为了确保所有消息能够在合理的时间内被读取，应用程序需要记录生成消息的数量（一般用每秒多少个消息来表示），而消费者需要记录已读取消息的数量（也用每秒多少个消息来表示）以及消息生成时间（生成消息的时间）到当前时间（读取消息的时间）之间的时间差。然后，你需要使用工具来比较生产者和消费者记录的消息数量（为了确保没有丢失消息），确保这两者之间的时间差不会超出我们允许的范围。为了做到更好的监控，我们可以增加一个“监控消费者”，这个消费者订阅一个特别的主题，它只进行消息的计数操作，并把数值与生成的消息数量进行对比，这样我们就可以在没有消费者的情况下仍然能够准确地监控生产者。这种端到端的监控系统实现起来很耗费时间，具有一定挑战性。据我们所知，目前还没有开源的实现。Confluent 提供了一个商业的实现版本，它是 Confluent Control Center 的一部分。

6.7　总结

正如我们在本章开头所说的，可靠性并不只是 Kafka 单方面的事情。我们应该从整个系统层面来考虑可靠性问题，包括应用程序的架构、生产者和消费者 API 的使用方式、生产者和消费者的配置、主题的配置以及 broker 的配置。系统的可靠性需要在许多方面作出权衡，比如复杂性、性能、可用性和磁盘空间的使用。掌握 Kafka 的各种配置和常用模式，对使用场景的需求做到心中有数，你就可以在应用程序和 Kafka 的可靠性程度以及各种权衡之间作出更好的选择。

第 7 章　构建数据管道

在使用 Kafka 构建数据管道时，通常有两种使用场景：第一种，把 Kafka 作为数据管道的两个端点之一，例如，把 Kafka 里的数据移动到 S3 上，或者把 MongoDB 里的数据移动到 Kafka 里；第二种，把 Kafka 作为数据管道两个端点的中间媒介，例如，为了把 Twitter 的数据移动到 ElasticSearch 上，需要先把它们移动到 Kafka 里，再将它们从 Kafka 移动到 ElasticSearch 上。

LinkedIn 和其他一些大公司都将 Kafka 用在上述两种场景中，后来，我们在 0.9 版本的 Kafka 里增加了 Kafka Connect（以下简称 Connect）。我们注意到，企业在将 Kafka 与数据管道进行集成时总

会碰到一些特定的问题，所以决定往 Kafka 里增加一些 API 来帮助他们解决这些问题，而不是等着他们提出这些问题。

Kafka 为数据管道带来的主要价值在于，它可以作为数据管道各个数据段之间的大型缓冲区，有效地解耦管道数据的生产者和消费者。Kafka 的解耦能力以及在安全和效率方面的可靠性，使它成为构建数据管道的最佳选择。

> 数据集成的场景
>
> 有些组织把 Kafka 看成是数据管道的一个端点，他们会想"我怎么才能把数据从 Kafka 移到 ElasticSearch 里"。这么想是理所当然的——特别是当你需要的数据在到达 ElasticSearch 之前还停留在 Kafka 里的时候，其实我们也是这么想的。不过我们要讨论的是如何在更大的场景里使用 Kafka，这些场景至少包含两个端点（可能会更多），而且这些端点都不是 Kafka。对于那些面临数据集成问题的人来说，我们建议他们从大局考虑问题，而不只是把注意力集中在少量的端点上。过度聚焦在短期问题上，只会增加后期维护的复杂性，付出更高的成本。

本章将讨论在构建数据管道时需要考虑的几个常见问题。这些问题并非 Kafka 独有，它们都是与数据集成相关的一般性问题。我们将解释为什么可以使用 Kafka 进行数据集成，以及它是如何解决这些问题的。我们将讨论 Connect API 与普通的客户端 API（Producer 和 Consumer）之间的区别，以及这些客户端 API 分别适合在什么情况下使用。然后我们会介绍 Connect。Connect 的完整手册不在本章的讨论范围之内，不过我们会举几个例子来帮助你入门，而且会告诉你可以从哪里了解到更多关于 Connect 的信息。最后介绍其他数据集成系统，以及如何将它们与 Kafka 集成起来。

7.1　构建数据管道时需要考虑的问题

本书不打算讲解所有有关构建数据管道的细节，我们会着重讨论在集成多个系统时需要考虑的几个最重要的问题。

7.1.1　及时性

有些系统希望每天一次性地接收大量数据，而有些则希望在数据生成

几毫秒之内就能拿到它们。大部分数据管道介于这两者之间。一个好的数据集成系统能够很好地支持数据管道的各种及时性需求，而且在业务需求发生变更时，具有不同及时性需求的数据表之间可以方便地进行迁移。Kafka 作为一个基于流的数据平台，提供了可靠且可伸缩的数据存储，可以支持几近实时的数据管道和基于小时的批处理。生产者可以频繁地向 Kafka 写入数据，也可以按需写入；消费者可以在数据到达的第一时间读取它们，也可以每隔一段时间读取一次积压的数据。

Kafka 在这里扮演了一个大型缓冲区的角色，降低了生产者和消费者之间的时间敏感度。实时的生产者和基于批处理的消费者可以同时存在，也可以任意组合。实现回压策略也因此变得更加容易，Kafka 本身就使用了回压策略（必要时可以延后向生产者发送确认），消费速率完全取决于消费者自己。

7.1.2　可靠性

我们要避免单点故障，并能够自动从各种故障中快速恢复。数据通过数据管道到达业务系统，哪怕出现几秒钟的故障，也会造成灾难性的影响，对于那些要求毫秒级的及时性系统来说尤为如此。数据传递保证是可靠性的另一个重要因素。有些系统允许数据丢失，不过在大多数情况下，它们要求至少一次传递。也就是说，源系统的每一个事件都必须到达目的地，不过有时候需要进行重试，而重试可能造成重复传递。有些系统甚至要求仅一次传递——源系统的每一个事件都必须到达目的地，不允许丢失，也不允许重复。

我们已经在第 6 章深入讨论了 Kafka 的可用性和可靠性保证。Kafka 本身就支持“至少一次传递”，如果再结合具有事务模型或唯一键特性的外部存储系统，Kafka 也能实现“仅一次传递”。因为大部分的端点都是数据存储系统，它们提供了“仅一次传递”的原语支持，所以基于 Kafka 的数据管道也能实现“仅一次传递”。值得一提的是，Connect API 为集成外部系统提供了处理偏移量的 API，连接器因此可以构建仅一次传递的端到端数据管道。实际上，很多开源的连接器都支持仅一次传递。

7.1.3　高吞吐量和动态吞吐量

为了满足现代数据系统的要求，数据管道需要支持非常高的吞吐量。更重要的是，在某些情况下，数据管道还需要能够应对突发的吞吐量增长。

由于我们将 Kafka 作为生产者和消费者之间的缓冲区，消费者的吞吐量和生产者的吞吐量就不会耦合在一起了。我们也不再需要实现复杂的回压机制，如果生产者的吞吐量超过了消费者的吞吐量，可以把数据积压在 Kafka 里，等待消费者追赶上来。通过增加额外的消费者或生产者可以实现 Kafka 的伸缩，因此我们可以在数据管道的任何一边进行动态的伸缩，以便满足持续变化的需求。

因为 Kafka 是一个高吞吐量的分布式系统，一个适当规模的集群每秒钟可以处理数百兆的数据，所以根本无需担心数据管道无法满足伸缩性需求。另外，Connect API 不仅支持伸缩，而且擅长并行处理任务。稍后，我们将会介绍数据源和数据池（Data Sink）如何在多个线程间拆分任务，最大限度地利用 CPU 资源，哪怕是运行在单台机器上。

Kafka 支持多种类型的压缩，在增长吞吐量时，Kafka 用户和管理员可以通过压缩来调整网络和存储资源的使用。

7.1.4　数据格式

数据管道需要协调各种数据格式和数据类型，这是数据管道的一个非常重要的因素。数据类型取决于不同的数据库和数据存储系统。你可能会通过 Avro 将 XML 或关系型数据加载到 Kafka 里，然后将它们转成 JSON 写入 ElasticSearch，或者转成 Parquet 写入 HDFS，或者转成 CSV 写入 S3。

Kafka 和 Connect API 与数据格式无关。我们已经在之前的章节介绍过，生产者和消费者可以使用各种序列化器来表示任意格式的数据。Connect API 有自己的内存对象模型，包括数据类型和 schema。不过，可以使用一些可插拔的转换器将这些对象保存成任意的格式，也就是说，不管数据是什么格式的，都不会限制我们使用连接器。

很多数据源和数据池都有 schema，我们从数据源读取 schema，把它们保存起来，并用它们验证数据格式的兼容性，甚至用它们更新数据池的 schema。从 MySQL 到 Hive 的数据管道就是一个很好的例子。如果有人在 MySQL 里增加了一个字段，那么在加载数据时，数据管道可以保证 Hive 里也添加了相应的字段。

另外，数据池连接器将 Kafka 的数据写入外部系统，因此需要负责处理数据格式。有些连接器把数据格式的处理做成可插拔的，比如 HDFS 的连接器就支持 Avro 和 Parquet。

通用的数据集成框架不仅要支持各种不同的数据类型，而且要处理好不同数据源和数据池之间的行为差异。例如，在关系型数据库向 Syslog 发起抓取数据请求时，Syslog 会将数据推送给它们，而 HDFS 只支持追加写入模式，只能向 HDFS 写入新数据，而对于其他很多系统来说，既可以追加数据，也可以更新已有的数据。

7.1.5 转换

数据转换比其他需求更具争议性。数据管道的构建可以分为两大阵营，即 ETL 和 ELT。 ETL 表示提取—转换—加载（Extract-Transform-Load），也就是说，当数据流经数据管道时，数据管道会负责处理它们。这种方式为我们节省了时间和存储空间，因为不需要经过保存数据、修改数据、再保存数据这样的过程。不过，这种好处也要视情况而定。有时候，这种方式会给我们带来实实在在的好处，但也有可能给数据管道造成不适当的计算和存储负担。这种方式有一个明显不足，就是数据的转换会给数据管道下游的应用造成一些限制，特别是当下游的应用希望对数据进行进一步处理的时候。假设有人在 MongoDB 和 MySQL 之间建立了数据管道，并且过滤掉了一些事件记录，或者移除了一些字段，那么下游应用从 MySQL 中访问到的数据是不完整的。如果它们想要访问被移除的字段，只能重新构建管道，并重新处理历史数据（如果可能的话）。

ELT 表示提取—加载—转换（Extract-Load-Transform）。在这种模式下，数据管道只做少量的转换（主要是数据类型转换），确保到达数据池的数据尽可能地与数据源保持一致。这种情况也被称为高保真（high fidelity）数据管道或数据湖（data lake）架构。目标系统收集“原始数据”，并负责处理它们。这种方式为目标系统的用户提供了最大的灵活性，因为它们可以访问到完整的数据。在这些系统里诊断问题也变得更加容易，因为数据被集中在同一个系统里进行处理，而不是分散在数据管道和其他应用里。这种方式的不足在于，数据的转换占用了目标系统太多的 CPU 和存储资源。有时候，目标系统造价高昂，如果有可能，人们希望能够将计算任务移出这些系统。

7.1.6 安全性

安全性是人们一直关心的问题。对于数据管道的安全性来说，人们主要关心如下几个方面。

- 我们能否保证流经数据管道的数据是经过加密的？这是跨数据中心数据管道通常需要考虑的一个主要方面。

- 谁能够修改数据管道？
- 如果数据管道需要从一个不受信任的位置读取或写入数据，是否有适当的认证机制？

Kafka 支持加密传输数据，从数据源到 Kafka，再从 Kafka 到数据池。它还支持认证（通过 SASL 来实现）和授权，所以你可以确信，如果一个主题包含了敏感信息，在不经授权的情况下，数据是不会流到不安全的系统里的。Kafka 还提供了审计日志用于跟踪访问记录。通过编写额外的代码，还可能跟踪到每个事件的来源和事件的修改者，从而在每个记录之间建立起整体的联系。

7.1.7　故障处理能力

我们不能总是假设数据是完美的，而要事先做好应对故障的准备。能否总是把缺损的数据挡在数据管道之外？能否恢复无法解析的记录？能否修复（或许可以手动进行）并重新处理缺损的数据？如果在若干天之后才发现原先看起来正常的数据其实是缺损数据，该怎么办？

因为 Kafka 会长时间地保留数据，所以我们可以在适当的时候回过头来重新处理出错的数据。

7.1.8　耦合性和灵活性

数据管道最重要的作用之一是解耦数据源和数据池。它们在很多情况下可能发生耦合。

临时数据管道

　　有些公司为每一对应用程序建立单独的数据管道。例如，他们使用 Logstash 向 ElasticSearch 导入日志，使用 Flume 向 HDFS 导入日志，使用 GoldenGate 将 Oracle 的数据导到 HDFS，使用 Informatica 将 MySQL 的数据或 XML 导到 Oracle，等等。他们将数据管道与特定的端点耦合起来，并创建了大量的集成点，需要额外的部署、维护和监控。当有新的系统加入时，他们需要构建额外的数据管道，从而增加了采用新技术的成本，同时遏制了创新。

元数据丢失

　　如果数据管道没有保留 schema 元数据，而且不允许 schema 发生变更，那么最终会导致生产者和消费者之间发生紧密的耦合。没有了 schema，生产者和消费者需要额外的信息来解析数据。假设数据

从 Oracle 流向 HDFS，如果 DBA 在 Oracle 里添加了一个字段，而且没有保留 schema 信息，也不允许修改 schema，那么从 HDFS 读取数据时可能会发生错误，因此需要双方的开发人员同时升级应用程序才能解决这个问题。不管是哪一种情况，它们的解决方案都不具备灵活性。如果数据管道允许 schema 发生变更，应用程序各方就可以修改自己的代码，无需担心对整个系统造成破坏。

末端处理

　　我们在讨论数据转换时就已提到，数据管道难免要做一些数据处理。在不同的系统之间移动数据肯定会碰到不同的数据格式和不同的应用场景。不过，如果数据管道过多地处理数据，那么就会给下游的系统造成一些限制。在构建数据管道时所做的设计决定都会对下游的系统造成束缚，比如应该保留哪些字段或应该如何聚合数据，等等。如果下游的系统有新的需求，那么数据管道就要作出相应的变更，这种方式不仅不灵活，而且低效、不安全。更为灵活的方式是尽量保留原始数据的完整性，让下游的应用自己决定如何处理和聚合数据。

7.2　如何在Connect API和客户端API之间作出选择

在向 Kafka 写入数据或从 Kafka 读取数据时，要么使用传统的生产者和消费者客户端，就像第 3 章和第 4 章所描述的那样，要么使用后面即将介绍的 Connect API 和连接器。在具体介绍 Connect API 之前，我们不妨先问自己一个问题：“什么时候适合用哪一个？”

我们知道，Kafka 客户端是要被内嵌到应用程序里的，应用程序使用它们向 Kafka 写入数据或从 Kafka 读取数据。如果你是开发人员，你会使用 Kafka 客户端将应用程序连接到 Kafka，并修改应用程序的代码，将数据推送到 Kafka 或者从 Kafka 读取数据。

如果要将 Kafka 连接到数据存储系统，可以使用 Connect，因为这些系统不是你开发的，你无法或者也不想修改它们的代码。Connect 可以用于从外部数据存储系统读取数据，或者将数据推送到外部存储系统。如果数据存储系统提供了相应的连接器，那么非开发人员就可以通过配置连接器的方式来使用 Connect。

如果你要连接的数据存储系统没有相应的连接器，那么可以考虑使用客户端 API 或 Connect API 开发一个应用程序。我们建议首选 Connect，因为它提供了一些开箱即用的特性，比如配置管理、偏移

量存储、并行处理、错误处理，而且支持多种数据类型和标准的 REST 管理 API。开发一个连接 Kafka 和外部数据存储系统的小应用程序看起来很简单，但其实还有很多细节需要处理，比如数据类型和配置选项，这些无疑加大了开发的复杂性——Connect 处理了大部分细节，让你可以专注于数据的传输。

7.3 Kafka Connect

Connect 是 Kafka 的一部分，它为在 Kafka 和外部数据存储系统之间移动数据提供了一种可靠且可伸缩的方式。它为连接器插件提供了一组 API 和一个运行时——Connect 负责运行这些插件，它们则负责移动数据。Connect 以 `worker` 进程集群的方式运行，我们基于 `worker` 进程安装连接器插件，然后使用 REST API 来管理和配置 `connector`，这些 `worker` 进程都是长时间持续运行的作业。连接器启动额外的 `task`，有效地利用工作节点的资源，以并行的方式移动大量的数据。数据源的连接器负责从源系统读取数据，并把数据对象提供给 `worker` 进程。数据池的连接器负责从 `worker` 进程获取数据，并把它们写入目标系统。Connect 通过 `connector` 在 Kafka 里存储不同格式的数据。Kafka 支持 JSON，而且 Confluent Schema Registry 提供了 Avro 转换器。开发人员可以选择数据的存储格式，这些完全独立于他们所使用的连接器。

本章的内容无法完全覆盖 Connect 的所有细节和各种连接器，这些内容可以单独写成一本书。不过，我们会提供 Connect 的概览，还会介绍如何使用它，并提供一些额外的参考资料。

7.3.1 运行Connect

Connect 随着 Kafka 一起发布，所以无需单独安装。如果你打算在生产环境使用 Connect 来移动大量的数据，或者打算运行多个连接器，那么最好把 Connect 部署在独立于 broker 的服务器上。在所有的机器上安装 Kafka，并在部分服务器上启动 broker，然后在其他服务器上启动 Connect。

启动 `Connect` 进程与启动 broker 差不多，在调用脚本时传入一个属性文件即可。

```
bin/connect-distributed.sh config/connect-distributed.properties
```

`Connect` 进程有以下几个重要的配置参数。

- `bootstrap.servers`：该参数列出了将要与 Connect 协同工作的 broker 服务器，连接器将会向这些 broker 写入数据或者从它们那里读取数据。你不需要指定集群所有的 broker，不过建议至少指定 3 个。
- `group.id`：具有相同 group id 的 worker 属于同一个 Connect 集群。集群的连接器和它们的任务可以运行在任意一个 worker 上。
- `key.converter` 和 `value.converter`：Connect 可以处理存储在 Kafka 里的不同格式的数据。这两个参数分别指定了消息的键和值所使用的转换器。默认使用 Kafka 提供的 JSONConverter，当然也可以配置成 Confluent Schema Registry 提供的 AvroConverter。

有些转换器还包含了特定的配置参数。例如，通过将 `key.converter.schema.enable` 设置成 `true` 或者 `false` 来指定 JSON 消息是否可以包含 schema。值转换器也有类似的配置，不过它的参数名是 `value.converter.schema.enable`。Avro 消息也包含了 schema，不过需要通过 `key.converter.schema.registry.url` 和 `value.converter.schema.registry.url` 来指定 Schema Registry 的位置。

我们一般通过 Connect 的 REST API 来配置和监控 `rest.host.name` 和 `rest.port` 连接器。你可以为 REST API 指定特定的端口。

在启动 worker 集群之后，可以通过 REST API 来验证它们是否运行正常。

```
gwen$ curl http://localhost:8083/
{"version":"0.10.1.0-SNAPSHOT","commit":"561f45d747cd2a8c"}
```

这个 REST URI 应该要返回当前 Connect 的版本号。我运行的是 Kafka 0.10.1.0（预发行）快照版本。我们还可以检查已经安装好的连接器插件：

```
gwen$ curl http://localhost:8083/connector-plugins

[{"class":"org.apache.kafka.connect.file.FileStreamSourceConnector"},
```

```
{"class":"org.apache.kafka.connect.file.FileStreamSinkConnector"}]
```

我运行的是最简单的 Kafka，所以只有文件数据源和文件数据池两种插件可用。

让我们先来看看如何配置和使用这些内置的连接器，然后再提供一些使用外部数据存储系统的高级示例。

单机模式

要注意，Connect 也支持单机模式。单机模式与分布式模式类似，只是在启动时使用 `bin/connect-standalone.sh` 代替 `bin/connect-distributed.sh`，也可以通过命令行传入连接器的配置文件，这样就不需要使用 REST API 了。在单机模式下，所有的连接器和任务都运行在单独的 worker 进程上。单机模式使用起来更简单，特别是在开发和诊断问题的时候，或者是在需要让连接器和任务运行在某台特定机器上的时候（比如 Syslog 连接器会监听某个端口，所以你需要知道它运行在哪台机器上）。

7.3.2 连接器示例——文件数据源和文件数据池

这个例子使用了文件连接器和 JSON 转换器，它们都是 Kafka 自带的。接下来要确保 Zookeeper 和 Kafka 都处于运行状态。

首先启动一个分布式的 worker 进程。为了实现高可用性，真实的生产环境一般需要至少 2~3 个 worker 集群。不过在这个例子里，我们只启动 1 个。

```
bin/connect-distributed.sh config/connect-distributed.properties &
```

现在开始启动一个文件数据源。为了方便，直接让它读取 Kafka 的配置文件——把 Kafka 的配置文件内容发送到主题上。

```
echo '{"name":"load-kafka-config", "config":{"connector.class":"FileStrea
Source","file":"config/server.properties","topic":"kafka-config-topic"}}'
curl -X POST -d @- http://localhost:8083/connectors --header "content-
Type:application/json"
```

```
{"name":"load-kafka-config","config":{"connector.class":"FileStream-
Source","file":"config/server.properties","topic":"kafka-configtopic","
name":"load-kafka-config"},"tasks":[]}
```

我写了一个 JSON 片段，里面包含了连接器的名字 `load-kafka-config` 和连接器的配置信息，配置信息包含了连接器的类名、需要加载的文件名和主题的名字。

下面通过 Kafka 的控制台消费者来验证配置文件是否已经被加载到主题上了：

```
gwen$ bin/kafka-console-consumer.sh --new --bootstrap-server=localhost:909
topic kafka-config-topic --from-beginning
```

如果一切正常，可以看到如下的输出：

```
{"schema":{"type":"string","optional":false},"payload":"# Licensed to the
Apache Software Foundation (ASF) under one or more"}

<省略部分>

{"schema":{"type":"string","optional":false},"payload":"##################
Server Basics
#############################"}
{"schema":{"type":"string","optional":false},"payload":""}
{"schema":{"type":"string","optional":false},"payload":"# The id of the br
This must be set to a unique integer for each broker."}
{"schema":{"type":"string","optional":false},"payload":"broker.id=0"}
{"schema":{"type":"string","optional":false},"payload":""}

<省略部分>
```

以上输出的是 `config/server.properties` 文件的内容，这些内容被一行一行地转成 JSON 记录，并被连接器发送到 `kafka-config-topic` 主题上。默认情况下，JSON 转换器会在每个记录里附带上 schema。这里的 schema 非常简单——只有一个 `payload` 列，它是字符串类型，并且包含了文件里的一行内容。

现在使用文件数据池的转换器把主题里的内容导到文件里。导出的文件内容应该与原始 `server.properties` 文件的内容完全一样，JSON 转化器将会把每个 JSON 记录转成单行文本。

```
echo '{"name":"dump-kafka-config", "config":
```

```
{"connector.class":"FileStreamSink","file":"copy-of-serverproperties","topics":"kafka-config-topic"}}' | curl -X POST -d @- http://localhost:8083/connectors --header "content-Type:application/json"

{"name":"dump-kafka-config","config":
{"connector.class":"FileStreamSink","file":"copy-of-serverproperties","topics":"kafka-config-topic","name":"dump-kafka-config"},"tasks":
[]}
```

这次的配置发生了变化：我们使用了类名 `FileStreamSink`，而不是`FileStreamSource`；文件属性指向目标文件，而不是原先的文件；我们指定了`topics`，而不是`topic`。可以使用数据池将多个主题写入一个文件，而一个数据源只允许被写入一个主题。

如果一切正常，你会得到一个叫作 `copy-of-server-properties` 的文件，该文件的内容与 `config/server.properties` 完全一样。

如果要删除一个连接器，可以运行下面的命令：

```
curl -X DELETE http://localhost:8083/connectors/dump-kafka-config
```

在删除连接器之后，如果查看 Connect 的日志，你会发现其他的连接器会重启它们的任务。这是为了在 worker 进程间平衡剩余的任务，确保删除连接器之后可以保持负载的均衡。

7.3.3　连接器示例——从MySQL到ElasticSearch

接下来，我们要做一些更有用的事情。这次，我们将一个 MySQL 的表数据导入到一个 Kafka 主题上，再将它们加载到 ElasticSearch 里，然后对它们的内容进行索引。

我是在自己的 Mac 笔记本上运行测试的，使用了下面的命令来安装 MySQL 和 ElasticSearch：

```
brew install mysql
brew install elasticsearch
```

下一步要确保已经有可用的连接器。如果使用的是 Confluent OpenSource，这个平台已经包含了相关的连接器，否则就要从

GitHub 上下载和安装。

(1) 打开网页 https://github.com/confluentinc/kafka-connect-elasticsearch。

(2) 把代码复制到本地。

(3) 使用 `mvn install` 来构建项目。

(4) 按照相同的步骤安装 JDBC 连接器：https://github.com/confluentinc/kafka-connect-jdbc。

接下来把每个 target 目录下生成的 JAR 包复制到 Connect 的类路径中。

```
gwen$ mkdir libs
gwen$ cp ../kafka-connect-jdbc/target/kafka-connect-jdbc-3.1.0-SNAPSHOT.ja
libs/
gwen$ cp ../kafka-connect-elasticsearch/target/kafka-connectelasticsearch-
3.2.0-SNAPSHOT-package/share/java/kafka-connect-elasticsearch/*
libs/
```

如果 worker 进程还没有启动，需要先启动它们，然后检查新的连接器插件是否已经安装成功：

```
gwen$  bin/connect-distributed.sh config/connect-distributed.properties &

gwen$  curl http://localhost:8083/connector-plugins
[{"class":"org.apache.kafka.connect.file.FileStreamSourceConnector"},
{"class":"io.confluent.connect.elasticsearch.ElasticsearchSinkConnector"},
{"class":"org.apache.kafka.connect.file.FileStreamSinkConnector"},
{"class":"io.confluent.connect.jdbc.JdbcSourceConnector"}]
```

从上面的输出可以看到，新的连接器插件已经安装成功了。JDBC 连接器还需要一个 MySQL 驱动程序。我们从 Oracle 网站下载了一个 MySQL 的 JDBC 驱动程序，并将其解压，然后把 `mysql-connector-java-5.1.40-bin.jar` 复制到 `libs/` 目录下。

下一步要在 MySQL 里创建一张表。可以使用 JDBC 连接器将其以流的方式发送给 Kafka。

```
gwen$ mysql.server restart
```

```
mysql> create database test;
Query OK, 1 row affected (0.00 sec)

mysql> use test;
Database changed
mysql> create table login (username varchar(30), login_time datetime);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into login values ('gwenshap', now());
Query OK, 1 row affected (0.01 sec)

mysql> insert into login values ('tpalino', now());
Query OK, 1 row affected (0.00 sec)

mysql> commit;
Query OK, 0 rows affected (0.01 sec)
```

我们创建了一个数据库和一张表，并插入了一些测试数据。

接下来要配置 JDBC 连接器。可以从文档中找到所有可用的配置项，也可以通过 REST API 找到它们：

```
gwen$ curl -X PUT -d "{}" localhost:8083/connector-plugins/JdbcSourceConne
config/validate --header "content-Type:application/json" | python -m json.

{
    "configs": [
        {
            "definition": {
            "default_value": "",
            "dependents": [],
            "display_name": "Timestamp Column Name",
            "documentation": "The name of the timestamp column to use
            to detect new or modified rows. This column may not be
            nullable.",
            "group": "Mode",
            "importance": "MEDIUM",
            "name": "timestamp.column.name",
            "order": 3,
            "required": false,
            "type": "STRING",
            "width": "MEDIUM"
        },
        <省略部分>
```

我们向 REST API 发起验证连接器的请求，并传给它一个空的配置，得到的是所有可用的配置项，并以 JSON 的格式返回。为了方便阅读，下面使用 Python 对 JSON 进行了格式化。

有了这些信息，就可以创建和配置 JDBC 连接器了：

```
echo '{"name":"mysql-login-connector", "config":{"connector.class":"JdbcSo
Connector","connection.url":"jdbc:mysql://127.0.0.1:3306/test?
user=root","mode":"timestamp","table.whitelist":"login","validate.
non.null":false,"timestamp.column.name":"login_time","topic.prefix":"
mysql."}}' | curl -X POST -d @- http://localhost:8083/connectors --header
"content-Type:application/json"

{"name":"mysql-login-connector","config":{"connector.class":"JdbcSourceCon
connection.url":"jdbc:mysql://127.0.0.1:3306/test?
user=root","mode":"timestamp","table.whitelist":"login","validate.non.null
se","timestamp.column.name":"login_time","topic.prefix":"mysql.","name":"m
connector"},"tasks":[]}
```

为了确保连接器工作正常，我们从 `mysql.login` 主题上读取数据。

```
gwen$ bin/kafka-console-consumer.sh --new --bootstrap-server=localhost:909
topic mysql.login --from-beginning

<省略部分>

{"schema":{"type":"struct","fields":
[{"type":"string","optional":true,"field":"username"},
{"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Time
version":1,"field":"login_time"}],"optional":false,"name":"login"},"payloa
username":"gwenshap","login_time":1476423962000}}
{"schema":{"type":"struct","fields":
[{"type":"string","optional":true,"field":"username"},
{"type":"int64","optional":true,"name":"org.apache.kafka.connect.data.Time
version":1,"field":"login_time"}],"optional":false,"name":"login"},"payloa
username":"tpalino","login_time":1476423981000}}
```

如果得到一个“主题不存在”的错误信息，或者看不到任何数据，可以检查一下 Connect 的日志。

```
[2016-10-16 19:39:40,482] ERROR Error while starting connector mysql-login
(org.apache.kafka.connect.runtime.WorkerConnector:108)
org.apache.kafka.connect.errors.ConnectException: java.sql.SQLException: A
denied for user 'root;'@'localhost' (using password: NO)
        at io.confluent.connect.jdbc.JdbcSourceConnector.start(JdbcSourceC
java:78)
```

我反复尝试了几次才找到正确的连接串。如果还有其他问题，请检查

类路径里是否包含了驱动程序，或者是否有数据表的读取权限。

你会看到，在连接器运行期间，向 login 表插入的数据会立即出现在 `mysql.login` 主题上。

把数据从 MySQL 移动到 Kafka 里就算完成了，接下来把数据从 Kafka 写到 ElasticSearch 里，这个会更有意思。

首先启动 ElasticSearch，并验证是否访问本地端口：

```
gwen$ elasticsearch &
gwen$ curl http://localhost:9200/
{
  "name" : "Hammerhead",
  "cluster_name" : "elasticsearch_gwen",
  "cluster_uuid" : "42D5GrxOQFebf83DYgNl-g",
  "version" : {
    "number" : "2.4.1",
    "build_hash" : "c67dc32e24162035d18d6fe1e952c4cbcbe79d16",
    "build_timestamp" : "2016-09-27T18:57:55Z",
    "build_snapshot" : false,
    "lucene_version" : "5.5.2"
  },
  "tagline" : "You Know, for Search"
}
```

下面启动连接器：

```
echo '{"name":"elastic-login-connector", "config":{"connector.class":"Elas
connection.url":"http://localhost:
9200","type.name":"mysql-data","topics":"mysql.login","key.ignore":true}}'
curl -X POST -d @- http://localhost:8083/connectors --header "content-
Type:application/json"

{"name":"elastic-login-connector","config":{"connector.class":"Elasticsear
SinkConnector","connection.url":"http://localhost:9200","type.name":"mysql
topics":"mysql.login","key.ignore":"true","name":"elastic-loginconnector"}
"tasks":[{"connector":"elastic-login-connector","task":0}]}
```

这里有一些配置项需要解释一下。`connection.url` 是本地 ElasticSearch 服务器的地址。默认情况下，每个 Kafka 主题对应 ElasticSearch 里的一个索引，主题的名字与索引的名字相同。我们需要在主题内为即将写入的数据定义好类别。我们假设所有数据属于同一种类别，所以硬编码了一个类别 `type.name=mysql-data`。只有 `mysql.login` 主题里的数据会被写入 ElasticSearch。另外，在

创建 MySQL 数据表时没有为其指定主键，而 Kafka 数据的键是 null，所以要让 ElasticSearch 连接器使用主题名字、分区 id 和偏移量作为数据的键。同时，需要把 `key.ignore` 设置为 `true`。

先来验证是否已经为 `mysql.login` 主题的数据创建好索引了。

```
gwen$ curl 'localhost:9200/_cat/indices?v'
health status index        pri rep docs.count docs.deleted store.size
pri.store.size
yellow open mysql.login      5   1          3            0     10.7kb
10.7kb
```

如果索引还没有创建好，可以检查一下 Connect 的日志。如果出现错误，一般都是因为缺少配置项或依赖包。如果一切正常，就可以查询到索引数据了。

```
gwen$ curl -s -X "GET" "http://localhost:9200/mysql.login/_search?pretty=t
{
  "took" : 29,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [ {
      "_index" : "mysql.login",
      "_type" : "mysql-data",
      "_id" : "mysql.login+0+1",
      "_score" : 1.0,
      "_source" : {
        "username" : "tpalino",
        "login_time" : 1476423981000
      }
    }, {
      "_index" : "mysql.login",
      "_type" : "mysql-data",
      "_id" : "mysql.login+0+2",
      "_score" : 1.0,
      "_source" : {
        "username" : "nnarkede",
        "login_time" : 1476672246000
      }
    }, {
      "_index" : "mysql.login",
      "_type" : "mysql-data",
```

```
      "_id" : "mysql.login+0+0",
      "_score" : 1.0,
      "_source" : {
        "username" : "gwenshap",
        "login_time" : 1476423962000
      }
    } ]
  }
}
```

如果在 MySQL 里插入新的数据，它们会自动出现在 Kafka 的 `mysql.login` 主题以及 ElasticSearch 相应的索引里。

现在，我们已经知道如何构建与安装 JDBC 连接器和 ElasticSearch 连接器了，也可以根据具体需要构建和安装任意的连接器。Confluent 提供了一个可用的连接器清单（http://www.confluent.io/product/connectors/），一些公司和社区在开发和维护这些连接器。你可以从中挑选你想用的连接器，从 GitHub 上获取它们的代码，自行构建，并根据文档或者通过 REST API 获取配置项，配置好以后在自己的 Connect 集群上运行。

构建自己的连接器

任何人都可以基于公开的 Connector API 创建自己的连接器。事实上，人们创建了各种连接器，然后把它们发布到连接器中心（Connector Hub），并告诉我们怎么使用它们。如果你在连接器中心找不到可以适配你要集成的数据存储系统的连接器，可以开发自己的连接器。你也可以把自己的连接器贡献给社区，让更多的人知道和使用它们。关于构建连接器的更多细节已经超出了本章的讨论范围，不过可以参考官方文档学习如何构建连接器（http://docs.confluent.io/3.0.1/connect/devguide.html）。我们也建议将已有的连接器作为入门参考，或者从使用 maven archtype（https://github.com/jcustenborder/kafka-connect-archtype）开始。另外，可以在 Kafka 的社区邮件组（users@kafka.apache.org）寻求帮助，或者在邮件组里展示自己的连接器。

7.3.4　深入理解Connect

要理解 Connect 的工作原理，需要先知道 3 个基本概念，以及它们之间是如何进行交互的。我们已经在之前的示例里演示了如何运行 **worker** 进程集群以及如何启动和关闭连接器。不过我们并没有深入解释转化器是如何处理数据的——转换器把 MySQL 的数据行转成 JSON 记录，然后由连接器将它们写入 Kafka。

现在让我们深入理解每一个组件，以及它们之间是如何进行交互的。

01. 连接器和任务

    连接器插件实现了 Connector API，API 包含了两部分内容。

    连接器

    连接器负责以下 3 件事情。

    - 决定需要运行多少个任务。
    - 按照任务来拆分数据复制。
    - 从 worker 进程获取任务配置并将其传递下去。例如，JDBC 连接器会连接到数据库，统计需要复制的数据表，并确定需要执行多少个任务，然后在配置参数 `max.tasks` 和实际数据量之间选择数值较小的那个作为任务数。在确定了任务数之后，连接器会为每个任务生成一个配置，配置里包含了连接器的配置项（比如 `connection.url`）和该任务需要复制的数据表。`taskConfigs()` 方法返回一个映射列表，这些映射包含了任务的相关配置。worker 进程负责启动和配置任务，每个任务只复制配置项里指定的数据表。如果通过 REST API 启动连接器，有可能会启动任意节点上的连接器，那么连接器的任务就会在该节点上执行。

    任务

    任务负责将数据移入或移出 Kafka。任务在初始化时会得到由 worker 进程分配的一个上下文：源系统上下文（Source Context）包含了一个对象，可以将源系统记录的偏移量保存在上下文里（例如，文件连接器的偏移量就是文件里的字节位置，JDBC 连接器的偏移量可以是数据表的主键 ID）。目标系统连接器的上下文提供了一些方法，连接器可以用它们操作从 Kafka 接收到的数据，比如进行数据清理、错误重试，或者将偏移量保存到外部系统以便实现仅一次传递。任务在完成初始

化之后，就开始按照连接器指定的配置（包含在一个 `Properties` 对象里）启动工作。源系统任务对外部系统进行轮询，并返回一些记录，worker 进程将这些记录发送到 Kafka。数据池任务通过 worker 进程接收来自 Kafka 的记录，并将它们写入外部系统。

05. **worker** 进程

worker 进程是连接器和任务的“容器”。它们负责处理 HTTP 请求，这些请求用于定义连接器和连接器的配置。它们还负责保存连接器的配置、启动连接器和连接器任务，并把配置信息传递给任务。如果一个 worker 进程停止工作或者发生崩溃，集群里的其他 worker 进程会感知到（Kafka 的消费者协议提供了心跳检测机制），并将崩溃进程的连接器和任务重新分配给其他进程。如果有新的进程加入集群，其他进程也会感知到，并将自己的连接器和任务分配给新的进程，确保工作负载的均衡。进程还负责提交偏移量，如果任务抛出异常，可以基于这些偏移量进行重试。

为了更好地理解 worker 进程，我们可以将其与连接器和任务进行简单的比较。连接器和任务负责“数据的移动”，而 worker 进程负责 REST API、配置管理、可靠性、高可用性、伸缩性和负载均衡。

这种关注点分离是 Connect API 给我们带来的最大好处，而这种好处是普通客户端 API 所不具备的。有经验的开发人员都知道，编写代码从 Kafka 读取数据并将其插入数据库只需要一到两天的时间，但是如果要处理好配置、异常、REST API、监控、部署、伸缩、失效等问题，可能需要几个月。如果你使用连接器来实现数据复制，连接器插件会为你处理掉一大堆复杂的问题。

06. 转化器和 **Connect** 的数据模型

数据模型和转化器是 Connect API 需要讨论的最后一部分内容。Connect 提供了一组数据 API——它们包含了数据对象和用于描述数据的 schema。例如，JDBC 连接器从数据库读取了一个字段，并基于这个字段的数据类型创建了一个 `Connect Schema` 对象。然后使用这些 Schema 对象创建一个包含了所有数据库字段的 `Struct`——我们保存了每一个字段的名字和它们的值。源连接器所做的事情都很相似——从源系统读取事

件，并为每个事件生成 schema 和值（值就是数据对象本身）。目标连接器正好相反，它们获取 schema 和值，并使用 schema 来解析值，然后写入到目标系统。

源连接器只负责基于 Data API 生成数据对象，那么 worker 进程是如何将这些数据对象保存到 Kafka 的？这个时候，转换器就派上用场了。用户在配置 worker 进程（或连接器）时可以选择使用合适的转化器，用于将数据保存到 Kafka。目前可用的转化器有 Avro、JSON 和 String。JSON 转化器可以在转换结果里附带上 schema，当然也可以不使用 schema，这个是可配的。Kafka 系统因此可以支持结构化的数据和半结构化的数据。连接器通过 Data API 将数据返回给 worker 进程，worker 进程使用指定的转化器将数据转换成 Avro 对象、 JSON 对象或者字符串，然后将它们写入 Kafka。

对于目标连接器来说，过程刚好相反——在从 Kafka 读取数据时，worker 进程使用指定的转换器将各种格式（Avro、JSON 或 String）的数据转换成 Data API 格式的对象，然后将它们传给目标连接器，目标连接器再将它们插入到目标系统。

Connect API 因此可以支持多种类型的数据，数据类型与连接器的实现是相互独立的——只要有可用的转换器，连接器和数据类型可以自由组合。

07. 偏移量管理

worker 进程的 REST API 提供了部署和配置管理服务，除此之外，worker 进程还提供了偏移量管理服务。连接器只要知道哪些数据是已经被处理过的，就可以通过 Kafka 提供的 API 来维护偏移量。

源连接器返回给 worker 进程的记录里包含了一个逻辑分区和一个逻辑偏移量。它们并非 Kafka 的分区和偏移量，而是源系统的分区和偏移量。例如，对于文件源来说，分区可以是一个文件，偏移量可以是文件里的一个行号或者字符号；而对于 JDBC 源来说，分区可以是一个数据表，偏移量可以是一条记录的主键。在设计一个源连接器时，要着重考虑如何对源系统的数据进行分区以及如何跟踪偏移量，这将影响连接器的并行能力，也决定了连接器是否能够实现至少一次传递或者仅一次传递。

源连接器返回的记录里包含了源系统的分区和偏移量，worker 进程将这些记录发送给 Kafka。如果 Kafka 确认记录保存成功，worker 进程就把偏移量保存下来。偏移量的存储机制是可插拔的，一般会使用 Kafka 主题来保存。如果连接器发生崩溃并重启，它可以从最近的偏移量继续处理数据。

目标连接器的处理过程恰好相反，不过也很相似。它们从 Kafka 上读取包含了主题、分区和偏移量信息的记录，然后调用连接器的 `put()` 方法，该方法会将记录保存到目标系统里。如果保存成功，连接器会通过消费者客户端将偏移量提交到 Kafka 上。

框架提供的偏移量跟踪机制简化了连接器的开发工作，并在使用多个连接器时保证了一定程度的行为一致性。

7.4 Connect之外的选择

现在，我们对 Connect API 有了更加深入的了解，不仅知道如何使用它们，还知道它们的一些工作原理。虽然 Connect API 为我们提供了便利和可靠性，但它并非唯一的选择。下面看看还有哪些可用的框架，以及在什么时候可以使用它们。

7.4.1 用于其他数据存储的摄入框架

虽然我们很想说 Kafka 是至高无上的明星，但肯定会有人不同意这种说法。有些人将 Hadoop 或 ElasticSearch 作为他们数据架构的基础，这些系统都有自己的数据摄入工具。Hadoop 使用了 Flume，ElasticSearch 使用了 Logstash 或 Fluentd。如果架构里包含了 Kafka，并且需要连接大量的源系统和目标系统，那么建议使用 Connect API 作为摄入工具。如果构建的系统是以 Hadoop 或 ElasticSearch 为中心的，Kafka 只是数据的来源之一，那么使用 Flume 或 Logstash 会更合适。

7.4.2 基于图形界面的ETL工具

从保守的 Informatica 到一些开源的替代方案，比如 Talend 和 Pentaho，或者更新的 Apache NiFi 和 StreamSets——这些 ETL 解决方案都支持将 Kafka 作为数据源和数据池。如果你已经使用了这些系统，比如 Pentaho，那么就可能不会为了 Kafka 而在系统里增加另一种集成工具。如果你已经习惯了基于图形界面的 ETL 数据管道解决方

案，那就继续使用它们。不过，这些系统有一些不足的地方，那就是它们的工作流比较复杂，如果你只是希望从 Kafka 里获取数据或者将数据写入 Kafka，那么它们就显得有点笨重。我们在本章的开头部分已经说过，在进行数据集成时，应该将注意力集中在消息的传输上。因此，对于我们来说，大部分 ETL 工具都太过复杂了。

我们极力建议将 Kafka 当成是一个支持数据集成（使用 Connect）、应用集成（使用生产者和消费者）和流式处理的平台。Kafka 完全可以成为 ETL 工具的替代品。

7.4.3　流式处理框架

几乎所有的流式处理框架都具备从 Kafka 读取数据并将数据写入外部系统的能力。如果你的目标系统支持流式处理，并且你已经打算使用流式框架处理来自 Kafka 的数据，那么使用相同的框架进行数据集成看起来是很合理的。这样可以省掉一个处理步骤（不需要保存来自 Kafka 的数据，而是直接从 Kafka 读取数据然后写到其他系统），不过在发生数据丢失或者出现脏数据时，诊断问题会变得很困难，因为这些框架并不知道数据是什么时候丢失的，或者什么时候出现了脏数据。

7.5　总结

本章讨论了如何使用 Kafka 进行数据集成，从解释为什么要使用 Kafka 进行数据集成开始，到说明数据集成方案的一般性考虑点。我们先解释了为什么 Kafka 和 Connect API 是一种更好的选择，然后给出了一些例子，演示如何在不同的场景下使用 Connect，并深入了解了 Connect 的工作原理，最后介绍了一些 Connect 之外的数据集成方案。

不管最终你选择了哪一种数据集成方案，都需要保证所有消息能够在各种恶劣条件下完成传递。我们相信，在与 Kafka 的可靠性特性结合起来之后，Connect 具有了极高的可靠性。不过，我们仍然需要对它们进行严格的测试，以确保你选择的数据集成系统能够在发生进程停止、机器崩溃、网络延迟和高负载的情况下不丢失消息。毕竟，数据集成系统应该只做一件事情，那就是传递数据。

可靠性是数据集成系统唯一一个重要的需求。在选择数据系统时，首先要明确需求（可以参考 7.1 节），并确保所选择的系统能够满足这些需求。除此之外，还要很好地了解数据集成方案，确保知道怎么使

用它们来满足需求。虽然 Kafka 支持至少一次传递的原语，但你也要小心谨慎，避免在配置上出现偏差，破坏了可靠性。

第 8 章　跨集群数据镜像

本书的大部分内容都是在讨论单个 Kafka 集群的配置、维护和使用。不过，在某些场景的架构里，可能需要用到多个集群。

有时候，这些集群相互独立，属于不同的部门，或者有不同的用途，那么就没有必要在集群间复制数据。有时候，因为对 SLA 有不同的要求，或者因为工作负载的不同，很难通过调整单个集群来满足所有的需求。还有一些时候，对安全有各种不同的要求。这些问题其实很容易解决，就是分别为它们创建不同的集群——管理多个集群本质上就是重复多次运行单独的集群。

在某些情况下，不同的集群之间相互依赖，管理员需要不停地在集群间复制数据。大部分数据库都支持复制（replication），也就是持续地在数据库服务器之间复制数据。不过，因为前面已经使用过“复制”这个词来描述在同一个集群的节点间移动数据，所以我们把集群间的数据复制叫作镜像（mirroring）。Kafka 内置的跨集群复制工具叫作 **MirrorMaker**。

在这一章，我们将讨论跨集群的数据镜像，它们既可以镜像所有数据，也可以镜像部分数据。我们先从一些常见的跨集群镜像场景开始，然后介绍这些场景所使用的架构模式，以及这些架构模式各自的优缺点。接下来我们会介绍 MirrorMaker 以及如何使用它，然后说明在进行部署和性能调优时需要注意的一些事项。最后我们会对 MirrorMaker 的一些替代方案进行比较。

8.1　跨集群镜像的使用场景

下面列出了几个使用跨集群镜像的场景。

区域集群和中心集群

　　有时候，一个公司会有多个数据中心，它们分布在不同的地理区域、不同的城市或不同的大洲。这些数据中心都有自己的 Kafka 集

群。有些应用程序只需要与本地的 Kafka 集群通信，而有些则需要访问多个数据中心的数据（否则就没必要考虑跨数据中心的复制方案了）。有很多情况需要跨数据中心，比如一个公司根据供需情况修改商品价格就是一个典型的场景。该公司在每个城市都有一个数据中心，它们收集所在城市的供需信息，并调整商品价格。这些信息将会被镜像到一个中心集群上，业务分析员就可以在上面生成整个公司的收益报告。

冗余（**DR**）

一个 Kafka 集群足以支撑所有的应用程序，不过你可能会担心集群因某些原因变得不可用，所以你希望有第二个 Kafka 集群，它与第一个集群有相同的数据，如果发生了紧急情况，可以将应用程序重定向到第二个集群上。

云迁移

现今有很多公司将它们的业务同时部署在本地数据中心和云端。为了实现冗余，应用程序通常会运行在云供应商的多个服务区域里，或者使用多个云服务。本地和每个云服务区域都会有一个 Kafka 集群。本地数据中心和云服务区域里的应用程序使用自己的 Kafka 集群，当然也会在数据中心之间传输数据。例如，如果云端部署了一个新的应用程序，它需要访问本地的数据。本地的应用程序负责更新数据，并把它们保存在本地的数据库里。我们可以使用 Connect 捕获这些数据库变更，并把它们保存到本地的 Kafka 集群里，然后再镜像到云端的 Kafka 集群上。这样有助于控制跨数据中心的流量成本，同时也有助于改进流量的监管和安全性。

8.2　多集群架构

现在，我们已经知道有哪些场景需要用到多个 Kafka 集群，接下来将介绍几种常见的架构模式。我们之前已经成功地基于这些模式实现了上述的几种场景。在讲解这些架构模式之前，先简单地介绍一下有关跨数据中心通信的现实情况。我们即将讨论的方案都是以特定的网络条件为前提的。

8.2.1　跨数据中心通信的一些现实情况

以下是在进行跨数据中心通信时需要考虑的一些问题。

高延迟

　　Kafka 集群之间的通信延迟随着集群间距离的增长而增加。虽然光缆的速度是恒定的，但集群间的网络跳转所带来的缓冲和堵塞会增加通信延迟。

有限的带宽

　　单个数据中心的广域网带宽远比我们想象的要低得多，而且可用的带宽时刻在发生变化。另外，高延迟让如何利用这些带宽变得更加困难。

高成本

　　不管你是在本地还是在云端运行 Kafka，集群之间的通信都需要更高的成本。部分原因是因为带宽有限，而增加带宽是很昂贵的，当然，这个与供应商制定的在数据中心、区域和云端之间传输数据的收费策略也有关系。

Kafka 服务器和客户端是按照单个数据中心进行设计、开发、测试和调优的。我们假设服务器和客户端之间具有很低的延迟和很高的带宽，在使用默认的超时时间和缓冲区大小时也是基于这个前提。因此，我们不建议跨多个数据中心安装 Kafka 服务器（不过稍后会介绍一些例外情况）。

大多数情况下，我们要避免向远程的数据中心生成数据，但如果这么做了，那么就要忍受高延迟，并且需要通过增加重试次数（LinkedIn 曾经为跨集群镜像设置了 32 000 多次重试次数）和增大缓冲区来解决潜在的网络分区问题（生产者和服务器之间临时断开连接）。

如果有了跨集群复制的需求，同时又禁用了从 broker 到 broker 之间的通信以及从生产者到 broker 之间的通信，那么我们必须允许从 broker 到消费者之间的通信。事实上，这是最安全的跨集群通信方式。在发生网络分区时，消费者无法从 Kafka 读取数据，数据会驻留在 Kafka 里，直到通信恢复正常。因此，网络分区不会造成任何数据丢失。不过，因为带宽有限，如果一个数据中心的多个应用程序需要从另一个数据中心的 Kafka 服务器上读取数据，我们倾向于为每一个数据中心安装一个 Kafka 集群，并在这些集群间复制数据，而不是让不同的应用程序通过广域网访问数据。

在讨论更多有关跨数据中心通信的调优策略之前，我们需要先知道以

下一些架构原则。

- 每个数据中心至少需要一个集群。
- 每两个数据中心之间的数据复制要做到每个事件仅复制一次（除非出现错误需要重试）。
- 如果有可能，尽量从远程数据中心读取数据，而不是向远程数据中心写入数据。

8.2.2 Hub和Spoke架构

这种架构适用于一个中心 Kafka 集群对应多个本地 Kafka 集群的情况，如图 8-1 所示。

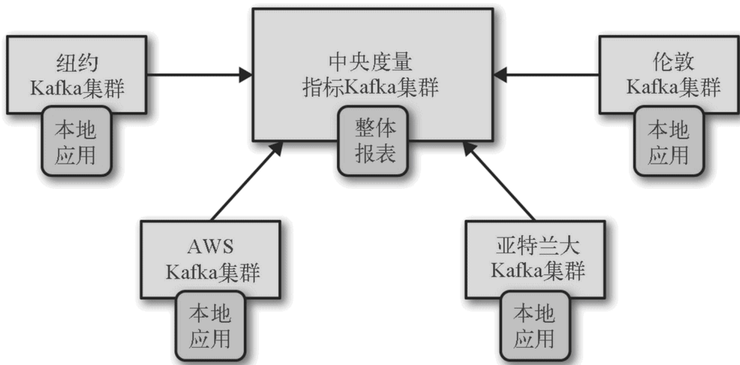


图 **8-1**：一个中心 **Kafka** 集群对应多个本地 **Kafka** 集群

这种架构有一个简单的变种，如果只有一个本地集群，那么整个架构里就只剩下两个集群：一个首领和一个跟随者，如图 8-2 所示。

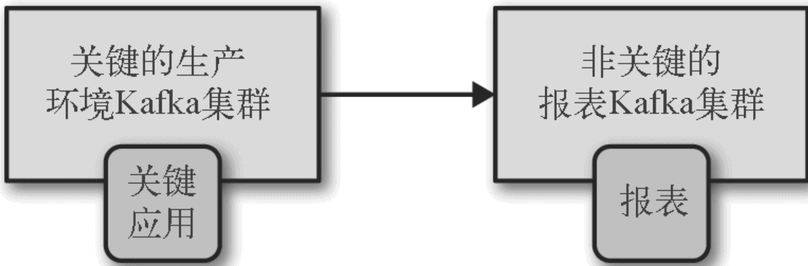


图 **8-2**：一个首领对应一个跟随者

当消费者需要访问的数据集分散在多个数据中心时，可以使用这种架

构。如果每个数据中心的应用程序只处理自己所在数据中心的数据，那么也可以使用这种架构，只不过它们无法访问到全局的数据集。

这种架构的好处在于，数据只会在本地的数据中心生成，而且每个数据中心的数据只会被镜像到中央数据中心一次。只处理单个数据中心数据的应用程序可以被部署在本地数据中心里，而需要处理多个数据中心数据的应用程序则需要被部署在中央数据中心里。因为数据复制是单向的，而且消费者总是从同一个集群读取数据，所以这种架构易于部署、配置和监控。

不过这种架构的简单性也导致了一些不足。一个数据中心的应用程序无法访问另一个数据中心的数据。为了更好地理解这种局限性，我们举一个例子来说明。

假设有一家银行，它在不同的城市有多家分行。每个城市的 Kafka 集群上保存了用户的信息和账号历史数据。我们把各个城市的数据复制到一个中心集群上，这样银行就可以利用这些数据进行业务分析。在用户访问银行网站或去他们所属的分行办理业务时，他们的请求被路由到本地集群上，同时从本地集群读取数据。假设一个用户去另一个城市的分行办理业务，因为他的信息不在这个城市，所以这个分行需要与远程的集群发生交互（不建议这么做），否则根本没有办法访问到这个用户的信息（很尴尬）。因此，这种架构模式在数据访问方面有所局限，因为区域数据中心之间的数据是完全独立的。

在采用这种架构时，每个区域数据中心的数据都需要被镜像到中央数据中心上。镜像进程会读取每一个区域数据中心的数据，并将它们重新生成到中心集群上。如果多个数据中心出现了重名的主题，那么这些主题的数据可以被写到中心集群的单个主题上，也可以被写到多个主题上。

8.2.3　双活架构

当有两个或多个数据中心需要共享数据并且每个数据中心都可以生产和读取数据时，可以使用双活（Active-Active）架构，如图 8-3 所示。

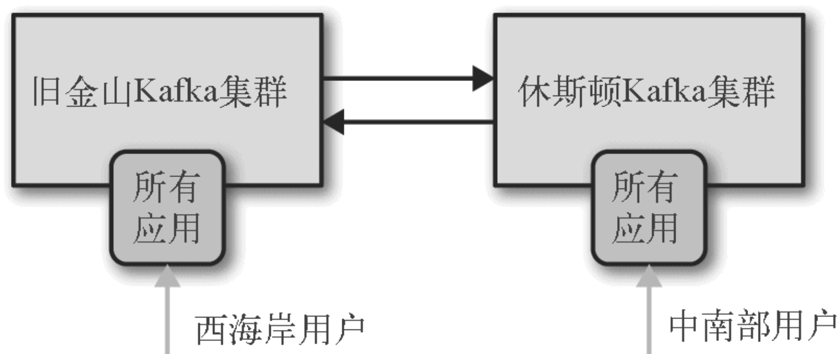


图 **8-3**：两个数据中心需要共享数据

这种架构的主要好处在于，它可以为就近的用户提供服务，具有性能上的优势，而且不会因为数据的可用性问题（在 Hub 和 Spoke 架构中就有这种问题）在功能方面作出牺牲。第二个好处是冗余和弹性。因为每个数据中心具备完整的功能，一旦一个数据中心发生失效，就可以把用户重定向到另一个数据中心。这种重定向完全是网络的重定向，因此是一种最简单、最透明的失效备援方案。

这种架构的主要问题在于，如何在进行多个位置的数据异步读取和异步更新时避免冲突。比如镜像技术方面的问题——如何确保同一个数据不会被无止境地来回镜像？而数据一致性方面的问题则更为关键。下面是可能遇到的问题。

- 如果用户向一个数据中心发送数据，同时从第二个数据中心读取数据，那么在用户读取数据之前，他发送的数据有可能还没有被镜像到第二个数据中心。对于用户来说，这就好比把一本书加入到购物车，但是在他点开购物车时，书却不在里面。因此，在使用这种架构时，开发人员经常会将用户“粘”在同一个数据中心上，以确保用户在大多数情况下使用的是同一个数据中心的数据（除非他们从远程进行连接或者数据中心不可用）。
- 一个用户在一个数据中心订购了书 A，而第二个数据中心几乎在同一时间收到了该用户订购书 B 的订单，在经过数据镜像之后，每个数据中心都包含了这两个事件。两个数据中心的应用程序需要知道如何处理这种情况。我们是否应该从中挑选一个作为“正确”的事件？如果是这样，我们需要在两个数据中心之间定义一致的规则，用于确定哪个事件才是正确的。又或者把两个都看成是正确的事件，将两本书都发给用户，然后设立一个部门专门来处理退货问题？ Amazon 就是使用这种方式来处

理冲突的，但对于股票交易部门来说，这种方案是行不通的。如何最小化冲突以及如何处理冲突要视具体情况而定。总之要记住，如果使用了这种架构，必然会遇到冲突问题，还要想办法解决它们。

如果能够很好地处理在从多个位置异步读取数据和异步更新数据时发生的冲突问题，那么我们强烈建议使用这种架构。这种架构是我们所知道的最具伸缩性、弹性、灵活性和成本优势的解决方案。所以，它值得我们投入精力去寻找一些办法，用于避免循环复制、把相同用户的请求粘在同一个数据中心，以及在发生冲突时解决冲突。

双活镜像（特别是当数据中心的数量超过两个）的挑战之处在于，每两个数据中心之间都需要进行镜像，而且是双向的。如果有 5 个数据中心，那么就需要维护至少 20 个镜像进程，还有可能达到 40 个，因为为了高可用，每个进程都需要冗余。

另外，我们还要避免循环镜像，相同的事件不能无止境地来回镜像。对于每一个“逻辑主题”，我们可以在每个数据中心里为它创建一个单独的主题，并确保不要从远程数据中心复制同名的主题。例如，对于逻辑主题“users”，我们在一个数据中心为其创建“SF.users”主题，在另一个数据中心为其创建“NYC.users”主题。镜像进程将 SF 的“SF.users”镜像到 NYC，同时将 NYC 的“NYC.users”镜像到 SF。这样一来，每一个事件只会被镜像一次，不过在经过镜像之后，每个数据中心同时拥有了 SF.users 和 NYC.users 这两个主题，也就是说，每个数据中心都拥有相同的用户数据。消费者如果要读取所有的用户数据，就需要以“*.users”的方式订阅主题。我们也可以把这种方式理解为数据中心的命名空间，比如在这个例子里，NYC 和 SF 就是命名空间。

在不久的将来，Kafka 将会增加记录头部信息。头部信息里可以包含源数据中心的信息，我们可以使用这些信息来避免循环镜像，也可以用它们来单独处理来自不同数据中心的数据。当然，你也可以通过使用结构化的数据格式（比如 Avro）来实现这一特性，并用它在数据里添加标签和头部信息。不过在进行镜像时，需要做一些额外的工作，因为现成的镜像工具并不支持自定义的头部信息格式。

8.2.4　主备架构

有时候，使用多个集群只是为了达到灾备的目的。你可能在同一个数据中心安装了两个集群，它们包含相同的数据，平常只使用其中的一个。当提供服务的集群完全不可用时，就可以使用第二个集群。又或

者你可能希望它们具备地理位置弹性，比如整体业务运行在加利福尼亚州的数据中心上，但需要在德克萨斯州有第二个数据中心，第二个数据中心平常不怎么用，但是一旦第一个数据中心发生地震，第二个数据中心就能派上用场。德克萨斯州的数据中心可能拥有所有应用程序和数据的非活跃（“冷”）复制，在紧急情况下，管理员可以启动它们，让第二个集群发挥作用。这种需求一般是合规性的，业务不一定会将其纳入规划范畴，但还是要做好充分的准备。主备（Active-Standby）架构示意图如图 8-4 所示。

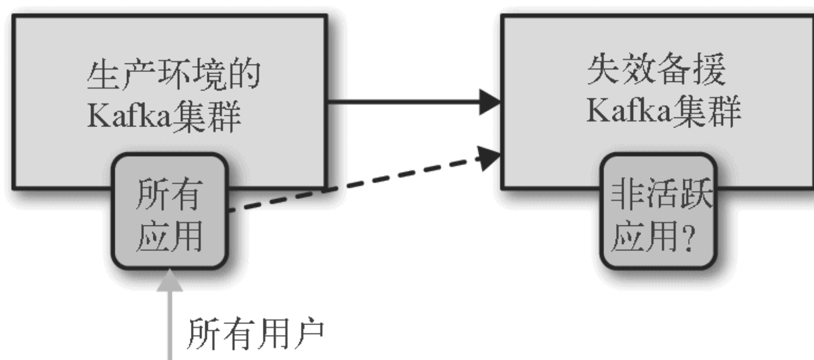


图 **8-4**：主备架构示意图

这种架构的好处是易于实现，而且可以被用于任何一种场景。你可以安装第二个集群，然后使用镜像进程将第一个集群的数据完整镜像到第二个集群上，不需要担心数据的访问和冲突问题，也不需要担心它会带来像其他架构那样的复杂性。

这种架构的不足在于，它浪费了一个集群。Kafka 集群间的失效备援比我们想象的要难得多。从目前的情况来看，要实现不丢失数据或无重复数据的 Kafka 集群失效备援是不可能的。我们只能尽量减少这些问题的发生，但无法完全避免。

让一个集群什么事也不做，只是等待灾难的发生，这明显就是对资源的浪费。因为灾难是（或者说应该是）很少见的，所以在大部分时间里，灾备集群什么事也不做。有些组织尝试减小灾备集群的规模，让它远小于生产环境的集群规模。这种做法具有一定的风险，因为你无法保证这种小规模的集群能够在紧急情况下发挥应有的作用。有些组织则倾向于让灾备集群在平常也能发挥作用，他们把一些只读的工作负载定向到灾备集群上，也就是说，实际上运行的是 Hub 和 Spoke 架构的一个简化版本，因为架构里只有一个 Spoke。

那么问题来了：如何实现 Kafka 集群的失效备援？

首先，不管选择哪一种失效备援方案，SRE（网站可靠性工程）团队都必须随时待命。今天能够正常运行的计划，在系统升级之后可能就无法正常工作，又或者已有的工具无法满足新场景的需求。每季度进行一次失效备援是最低限度的要求，一个高效的 SRE 团队会更频繁地进行失效备援。Chaos Monkey 是 Netflix 提供的一个著名的服务，它随机地制造灾难，有可能让任何一天都成为失效备援日。

现在，让我们来看看失效备援都包括哪些内容。

01. 数据丢失和不一致性

    因为 Kafka 的各种镜像解决方案都是异步的（8.2.5 节将介绍一种同步的方案），所以灾备集群总是无法及时地获取主集群的最新数据。我们要时刻注意灾备集群与主集群之间拉开了多少距离，并保证不要出现太大的差距。不过，一个繁忙的系统可以允许灾备集群与主集群之间有几百个甚至几千个消息的延迟。如果你的 Kafka 集群每秒钟可以处理 100 万个消息，而在主集群和灾备集群之间有 5ms 的延迟，那么在最好的情况下，灾备集群每秒钟会有 5000 个消息的延迟。所以，不在计划内的失效备援会造成数据的丢失。在进行计划内的失效备援时，可以先停止主集群，等待镜像进程将剩余的数据镜像完毕，然后切换到灾备集群，这样可以避免数据丢失。在发生非计划内的失效备援时，可能会丢失数千个消息。目前 Kafka 还不支持事务，也就是说，如果多个主题的数据（比如销售数据和产品数据）之间有相关性，那么在失效备援过程中，一些数据可以及时到达灾备集群，而有些则不能。那么在切换到灾备集群之后，应用程序需要知道该如何处理没有相关销售信息的产品数据。

02. 失效备援之后的起始偏移量

    在切换到灾备集群的过程中，最具挑战性的事情莫过于如何让应用程序知道该从什么地方开始继续处理数据。下面将介绍一些常用的方法，其中有些很简单，但有可能会造成额外的数据丢失或数据重复；有些则比较复杂，但可以最小化丢失数据和出现重复数据的可能性。

    偏移量自动重置

Kafka 消费者有一个配置选项，用于指定在没有上一个提交偏移量的情况下该作何处理。消费者要么从分区的起始位置开始读取数据，要么从分区的末尾开始读取数据。如果使用的是旧版本的消费者（偏移量保存在 Zookeeper 上），而且因为某些原因，这些偏移量没有被纳入灾备计划，那么就需要从上述两个选项中选择一个。要么从头开始读取数据，并处理大量的重复数据，要么直接跳到末尾，放弃一些数据（希望只是少量的数据）。如果重复处理数据或者丢失一些数据不会造成太大问题，那么重置偏移量是最为简单的方案。不过直接从主题的末尾开始读取数据这种方式或许更为常见。

复制偏移量主题

如果使用新的 Kafka 消费者（0.9 或以上版本），消费者会把偏移量提交到一个叫作 _consumer_offsets 的主题上。如果对这个主题进行了镜像，那么当消费者开始读取灾备集群的数据时，它们就可以从原先的偏移量位置开始处理数据。这个看起来很简单，不过仍然有很多需要注意的事项。

首先，我们并不能保证主集群里的偏移量与灾备集群里的偏移量是完全匹配的。假设主集群里的数据只保留 3 天，而你在一个星期之后才开始镜像，那么在这种情况下，主集群里第一个可用的偏移量可能是 57 000 000（前 4 天的旧数据已经被删除了），而灾备集群里的第一个偏移量是 0，那么当消费者尝试从 57 000 003 处（因为这是它要读取的下一个数据）开始读取数据时，就会失败。

其次，就算在主题创建之后立即开始镜像，让主集群和灾备集群的主题偏移量都从 0 开始，生产者在后续进行重试时仍然会造成偏移量的偏离。简而言之，目前的 Kafka 镜像解决方案无法为主集群和灾备集群保留偏移量。

最后，就算偏移量被完美地保留下来，因为主集群和灾备集群之间的延迟以及 Kafka 缺乏对事务的支持，消费者提交的偏移量有可能会在记录之前或者记录之后到达。在发生失效备援之后，消费者可能会发现偏移量与记录不匹配，或者灾备集群里最新的偏移量比主集群里的最新偏移量小。如图 8-5 所示。

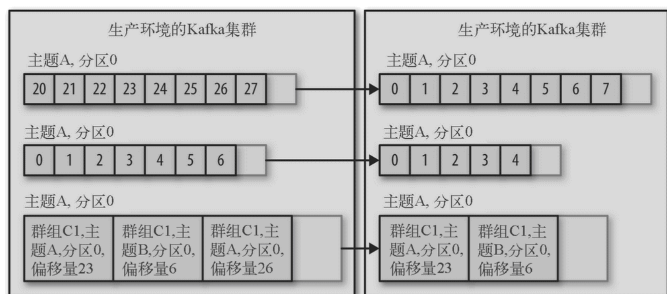


图 **8-5**：灾备集群偏移量与主集群的最新偏移量不匹配的示例

在这些情况下，我们需要接受一定程度的重复数据。如果灾备集群最新的偏移量比主集群的最新偏移量小，或者因为生产者进行重试导致灾备集群的记录偏移量比主集群的记录偏移量大，都会造成数据重复。你还需要知道该怎么处理最新偏移量与记录不匹配的问题，此时要从主题的起始位置开始读取还是从末尾开始读取？

复制偏移量主题的方式可以用于减少数据重复或数据丢失，而且实现起来很简单，只要及时地从 0 开始镜像数据，并持续地镜像偏移量主题就可以了。不过一定要注意上述的几个问题。

基于时间的失效备援

　　如果使用的是新版本（0.10.0 及以上版本）的 Kafka 消费者，每个消息里都包含了一个时间戳，这个时间戳指明了消息发送给 Kafka 的时间。在更新版本的 Kafka（0.10.1.0 及以上版本）里，broker 提供了一个索引和一个 API，用于根据时间戳查找偏移量。于是，假设你正在进行失效备援，并且知道失效事件发生在凌晨 4:05，那么就可以让消费者从 4:03 的位置开始处理数据。在两分钟的时间差里会存在一些重复数据，不过这种方式仍然比其他方案要好得多，而且也很容易向其他人解释——“我们将从凌晨 4:03 的位置开始处理数据”这样的解释要比“我们从一个不知道是不是最新的位置开始处理数据”要好得多。所以，这是一种更好的折中。问题是，如何让消费者从凌晨 4:03 的位置开始处理数据呢？

　　可以让应用程序来完成这件事情。我们为用户提供一个配置参数，用于指定从什么时间点开始处理数据。如果用户指定了时间，应用程序可以通过新的 API 获取指定时间的偏移量，

然后从这个位置开始处理数据。

如果应用程序在一开始就是这么设计的，那么使用这种方案就再好不过了。但如果应用程序在一开始不是这么设计的呢？开发一个这样的小工具也并不难——接收一个时间戳，使用新的 API 获取相应的偏移量，然后提交偏移量。我们希望在未来的 Kafka 版本里添加这样的工具，不过你也可以自己写一个。在运行这个工具时，应该先关闭消费者群组，在工具完成任务之后再启动它们。

该方案适用于那些使用了新版 Kafka、对失效备援有明确要求并且喜欢自己开发工具的人。

偏移量外部映射

我们知道，镜像偏移量主题的一个最大问题在于主集群和灾备集群的偏移量会发生偏差。因此，一些组织选择使用外部数据存储（比如 Apache Cassandra）来保存集群之间的偏移量映射。他们自己开发镜像工具，在一个数据被镜像到灾备集群之后，主集群和灾备集群的偏移量被保存到外部数据存储上。或者只有当两边的偏移量差值发生变化时，才保存这两个偏移量。比如，主集群的偏移量 495 被映射到灾备集群的偏移量 500，在外部存储上记录为（495,500）。如果之后因为消息重复导致差值发生变化，偏移量 596 被映射为 600，那么就保留新的映射（569,600）。他们没有必要保留 495 和 596 之间的所有偏移量映射，他们假设差值都是一样的，所以主集群的偏移量 550 会映射到灾备集群的偏移量 555。那么在发生失效备援时，他们将主集群的偏移量与灾备集群的偏移量映射起来，而不是在时间戳（通常会有点不准确）和偏移量之间做映射。他们通过上述技术手段之一来强制消费者使用映射当中的偏移量。对于那些在数据记录之前达到的偏移量或者没有及时被镜像到灾备集群的偏移量来说，仍然会有问题——不过这至少已经满足了部分场景的需求。

这种方案非常复杂，我认为并不值得投入额外的时间。在索引还没有出现之前，或许可以考虑使用这种方案。但在今天，我倾向于将集群升级到新版本，并使用基于时间戳的解决方案，而不是进行偏移量映射，更何况偏移量映射并不能覆盖所有的失效备援场景。

03. 在失效备援之后

假设失效备援进行得很顺利，灾备集群也运行得很正常，现在需要对主集群做一些改动，比如把它变成灾备集群。

如果能够通过简单地改变镜像进程的方向，让它将数据从新的主集群镜像到旧的主集群上面，事情就完美了！不过，这里还存在两个问题。

- 怎么知道该从哪里开始镜像？我们同样需要解决与镜像程序里的消费者相关的问题。而且不要忘了，所有的解决方案都有可能出现重复数据或者丢失数据，或者两者兼有。
- 之前讨论过，旧的主集群可能会有一些数据没有被镜像到灾备集群上，如果在这个时候把新的数据镜像回来，那么历史遗留数据还会继续存在，两个集群的数据就会出现不一致。

基于上述的考虑，最简单的解决方案是清理旧的主集群，删掉所有的数据和偏移量，然后从新的主集群上把数据镜像回来，这样可以保证两个集群的数据是一致的。

06. 关于集群发现

在设计灾备集群时，需要考虑一个很重要的问题，就是在发生失效备援之后，应用程序需要知道如何与灾备集群发起通信。不建议把主集群的主机地址硬编码在生产者和消费者的配置属性文件里。大多数组织为此创建了 DNS 别名，将其指向主集群，一旦发生紧急情况，可以将其指向灾备集群。有些组织则使用服务发现工具，比如 Zookeeper、Etcd 或 Consul。这些服务发现工具（DNS 或其他）没有必要将所有 broker 的信息都包含在内， Kafka 客户端只需要连接到其中的一个 broker，就可以获取到整个集群的元数据，并发现集群里的其他 broker。一般提供 3 个 broker 的信息就可以了。除了服务发现之外，在大多数情况下，需要重启消费者应用程序，这样它们才能找到新的可用偏移量，然后继续读取数据。

8.2.5　延展集群

在主备架构里，当 Kafka 集群发生失效时，可以将应用程序重定向到另一个集群上，以保证业务的正常运行。而在整个数据中心发生故障时，可以使用延展集群（stretch cluster）来避免 Kafka 集群失效。延展集群就是跨多个数据中心安装的单个 Kafka 集群。

延展集群与其他类型的集群有本质上的区别。首先，延展集群并非多个集群，而是单个集群，因此不需要对延展集群进行镜像。延展集群使用 Kafka 内置的复制机制在集群的 broker 之间同步数据。我们可以通过配置打开延展集群的同步复制功能，生产者会在消息成功写入到其他数据中心之后收到确认。同步复制功能要求使用机架信息，确保每个分区在其他数据中心都存在副本，还需要配置 `min.isr` 和 `acks=all`，确保每次写入消息时都可以收到至少两个数据中心的确认。

同步复制是这种架构的最大优势。有些类型的业务要求灾备站点与主站点保持 100% 的同步，这是一种合规性需求，可以应用在公司的任何一个数据存储上，包括 Kafka 本身。这种架构的另一个好处是，数据中心及所有 broker 都发挥了作用，不存在像主备架构那样的资源浪费。

这种架构的不足之处在于，它所能应对的灾难类型很有限，只能应对数据中心的故障，无法应对应用程序或者 Kafka 故障。运维的复杂性是它的另一个不足之处，它所需要的物理基础设施并不是所有公司都能够承担得起的。

如果能够在至少 3 个具有高带宽和低延迟的数据中心上安装 Kafka（包括 Zookeeper），那么就可以使用这种架构。如果你的公司有 3 栋大楼处于同一个街区，或者你的云供应商在同一个地区有 3 个可用的区域，那么就可以考虑使用这种方案。

为什么是 3 个数据中心？主要是因为 Zookeeper。Zookeeper 要求集群里的节点个数是奇数，而且只有当大多数节点可用时，整个集群才可用。如果只有两个数据中心和奇数个节点，那么其中的一个数据中心将包含大多数节点，也就是说，如果这个数据中心不可用，那么 Zookeeper 和 Kafka 也不可用。如果有 3 个数据中心，那么在分配节点时，可以做到每个数据中心都不会包含大多数节点。如果其中的一个数据中心不可用，其他两个数据中心包含了大多数节点，此时 Zookeeper 和 Kafka 仍然可用。

从理论上说，在两个数据中心运行 Zookeeper 和 Kafka 是可能的，只要将 Zookeeper 的群组配置成允许手动进行失效备援。不过在实际应用当中，这种做法并不常见。

8.3　Kafka的MirrorMaker

Kafka 提供了一个简单的工具，用于在两个数据中心之间镜像数据。这个工具叫 MirrorMaker，它包含了一组消费者（因为历史原因，它们在 MirrorMaker 文档里被称为 流），这些消费者属于同一个群组，并从主题上读取数据。每个 MirrorMaker 进程都有一个单独的生产者。镜像过程很简单：MirrorMaker 为每个消费者分配一个线程，消费者从源集群的主题和分区上读取数据，然后通过公共生产者将数据发送到目标集群上。默认情况下，消费者每 60 秒通知生产者发送所有的数据到 Kafka，并等待 Kafka 的确认。然后消费者再通知源集群提交这些事件相应的偏移量。这样可以保证不丢失数据（在源集群提交偏移量之前，Kafka 对消息进行了确认），而且如果 MirrorMaker 进程发生崩溃，最多只会出现 60 秒的重复数据。见图 8-6。

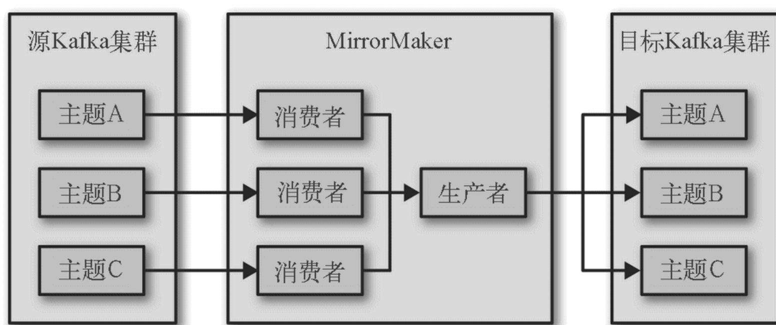


图 **8-6**：**MirrorMaker** 的镜像过程

> **MirrorMaker** 相关信息
>
> MirrorMaker 看起来很简单，不过出于对效率的考虑，以及尽可能地做到仅一次传递，它的实现并不容易。截止到 Kafka 0.10.0.0 版本，MirrorMaker 已经被重写了 4 次，而且在未来有可能会进行更多的重写。这里所描述的以及后续章节将提及的 MirrorMaker 相关细节都基于 0.9.0.0 到 0.10.2.0 之间的版本。

8.3.1　如何配置

MirrorMaker 是高度可配置的。首先，它使用了一个生产者和多个消费者，所以生产者和消费者的相关配置参数都可以用于配置 MirrorMaker。另外，MirrorMaker 本身也有一些配置参数，这些配

置参数之间有时候会有比较复杂的依赖关系。下面将举一些例子，并着重说明一些重要的配置参数。不过，MirrorMaker 的详细文档不在本书的讨论范围之内。

先来看一个 MirrorMaker 的例子：

```
bin/kafka-mirror-maker --consumer.config etc/kafka/consumer.properties --
producer.config etc/kafka/producer.properties --new.consumer --num.streams
whitelist ".*"
```

接下来分别说明 MirrorMaker 的基本命令行参数。

consumer.config

该参数用于指定消费者的配置文件。所有的消费者将共用这个配置，也就是说，只能配置一个源集群和一个 group.id。所有的消费者属于同一个消费者群组，这正好与我们的要求不谋而合。配置文件里有两个必选的参数：bootstrap.servers（源集群的服务器地址）和 group.id。除了这两个参数外，还可以为消费者指定其他任意的配置参数。 auto.commit.enable 参数一般不需要修改，用默认值 false 就行。MirrorMaker 会在消息安全到达目标集群之后提交偏移量，所以不能使用自动提交。如果修改了这个参数，可能会导致数据丢失。auto.offset.reset 参数一般需要进行修改，默认值是 latest，也就是说，MirrorMaker 只对那些在 MirrorMaker 启动之后到达源集群的数据进行镜像。如果想要镜像之前的数据，需要把该参数设为 earliest。我们将在 8.3.3 节介绍更多的配置属性。

producer.config

该参数用于指定生产者的配置文件。配置文件里唯一必选的参数是 bootstrap.servers（目标集群的服务器地址）。我们将在 8.3.3 节介绍更多的配置属性。

new.consumer

MirrorMaker 只能使用 0.8 版本或者 0.9 版本的消费者。建议使用 0.9 版本的消费者，因为它更加稳定。

num.streams

之前已经解释过，一个流就是一个消费者。所有的消费者共用一个生产者，MirrorMaker 将会使用这些流来填充同一个生产者。如果需要额外的吞吐量，就需要创建另一个 MirrorMaker 进程。

`whitelist`

这是一个正则表达式，代表了需要进行镜像的主题名字。所有与表达式匹配的主题都将被镜像。在这个例子里，我们希望镜像所有的主题，不过在实际当中最好使用类似“prod.*”这样的表达式，避免镜像测试用的主题。在双活架构中，MirrorMaker 将 NYC 数据中心的数据镜像到 SF，为其配置了 `whilelist="NYC.\*"`，这样就不会将 SF 的主题重新镜像回来。

8.3.2 在生产环境部署MirrorMaker

在上面的例子里，我们是从命令行启动 MirrorMaker 的。在生产环境，MirrorMaker 一般是作为后台服务运行的，而且是以 `nohup` 的方式运行，并将控制台的输出重定向到一个日志文件里。这个工具有一个 `-deamon` 命令行参数。理论上，只要使用这个参数就能实现后台运行，不需要再做其他任何事情，但在实际当中，最近发布的一些版本并不能如我们所期望的那样。

大部分使用 MirrorMaker 的公司都有自己的启动脚本，他们一般会使用部署系统（比如 Ansible、Puppet、Chef 和 Salt）实现自动化的部署和配置管理。

在 Docker 容器里运行 MirrorMaker 变得越来越流行。MirrorMaker 是完全无状态的，也不需要磁盘存储（所有的数据和状态都保存在 Kafka 上）。将 MirrorMaker 安装在 Docker 里，就可以实现在单台主机上运行多个 MirrorMaker 实例。因为单个 MirrorMaker 实例的吞吐量受限于单个生产者，所以为了提升吞吐量，需要运行多个 MirrorMaker 实例，而 Docker 简化了这一过程。Docker 也让 MirrorMaker 的伸缩变得更加容易，在流量高峰时，可以通过增加更多的容器来提升吞吐量，在流量低谷时，则减少容器。如果在云端运行 MirrorMaker，根据吞吐量实际情况，可以通过增加额外的服务器来运行 Docker 容器。

如果有可能，尽量让 MirrorMaker 运行在目标数据中心里。也就是说，如果要将 NYC 的数据发送到 SF，MirrorMaker 应该运行在 SF 的数据中心里。因为长距离的外部网络比数据中心的内部网络更加不可靠，如果发生了网络分区，数据中心之间断开了连接，那么一个无

法连接到集群的消费者要比一个无法连接到集群的生产者要安全得多。如果消费者无法连接到集群，最多也就是无法读取数据，数据仍然会在 Kafka 集群里保留很长的一段时间，不会有丢失的风险。相反，在发生网络分区时，如果 MirrorMaker 已经读取了数据，但无法将数据生成到目标集群上，就会造成数据丢失。所以说，远程读取比远程生成更加安全。

那么，什么情况下需要在本地读取消息并将其生成到远程数据中心呢？如果需要加密传输数据，但又不想在数据中心进行加密，就可以使用这种方式。消费者通过 SSL 连接到 Kafka 对性能有一定的影响，这个比生产者要严重得多，而且这种性能问题也会影响 broker。如果跨数据中心流量需要加密，那么最好把 MirrorMaker 放在源数据中心，让它读取本地的非加密数据，然后通过 SSL 连接将数据生成到远程的数据中心。这个时候，使用 SSL 连接的是生产者，所以性能问题就不那么明显了。在使用这种方式时，需要确保 MirrorMaker 在收到目标 broker 副本的有效确认之前不要提交偏移量，并在重试次数超出限制或者生产者缓冲区溢出的情况下立即停止镜像。

如果希望减小源集群和目标集群之间的延迟，可以在不同的机器上运行至少两个 MirrorMaker 实例，而且它们要使用相同的消费者群组。也就是说，如果关掉其中一台服务器，另一个 MirrorMaker 实例能够继续镜像数据。

在将 MirrorMaker 部署到生产环境时，最好要对以下几项内容进行监控。

延迟监控

　　我们绝对有必要知道目标集群是否落后于源集群。延迟体现在源集群最新偏移量和目标集群最新偏移量的差异上。见图 8-7。

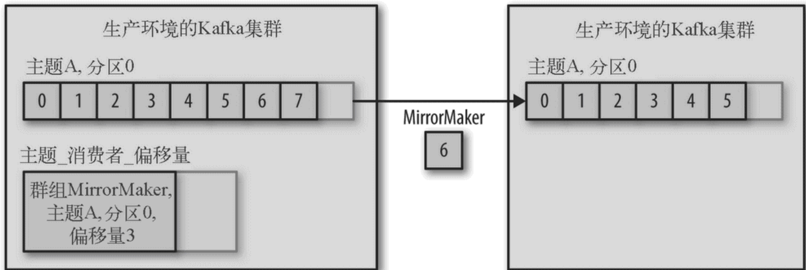


图 **8-7**：监控不同偏移量之间的延迟

如图 8-7 所示，源集群的最后一个偏移量是 7，而目标集群的最后一个偏移量是 5，所以它们之间有两个消息的延迟。

有两种方式可用于跟踪延迟，不过它们都不是完美的解决方案。

- 检查 MirrorMaker 提交到源集群的最新偏移量。可以使用 kafka-consumer-groups 工具检查 MirrorMaker 读取的每一个分区，查看分区的最新偏移量，也就是 MirrorMaker 提交的最新偏移量。不过这个偏移量并不会 100% 的准确，因为 MirrorMaker 并不会每时每刻都提交偏移量，默认情况下，它会每分钟提交一次。所以，我们最多会看到一分钟的延迟，然后延迟突然下降。图 8-7 中的延迟是 2，但 kafka-consumer-groups 会认为是 4，因为 MirrorMaker 还没有提交最近的偏移量。LinkedIn 的 burrow 也会监控这些信息，不过它使用了更为复杂的方法来识别延迟的真实性，所以不会导致误报。
- 检查 MirrorMaker 读取的最新偏移量（即使还未提交）。消费者通过 JMX 发布关键性度量指标，其中有一个指标是指消费者的最大延迟（基于它所读取的所有分区计算得出的）。这个延迟也不是 100% 的准确，因为它只反映了消费者读取的数据，并没有考虑生产者是否成功地将数据发送到目标集群上。在图 8-7 的示例里，MirrorMaker 消费者会认为延迟是 1，而不是 2，因为它已经读取了消息 6，尽管这个消息还没有被生成到目标集群上。

要注意，如果 MirrorMaker 跳过或丢弃部分消息，上述的两种方法是无法检测到的，因为它们只跟踪最新的偏移量。Confluent 的 Control Center 通过监控消息的数量和校验和来提升监控的准确性。

度量指标监控

MirrorMaker 内嵌了生产者和消费者，它们都有很多可用的度量指标，所以建议对它们进行监控。Kafka 文档列出了所有可用的度量指标。下面列出了几个已经被证明能够提升 MirrorMaker 性能的度量指标。

消费者

`fetch-size-avg`、`fetch-size-max`、`fetch-rate`、`fetch-throttle-time-avg` 以及 `fetch-throttle-time-max`。

生产者

batch-size-avg、batch-size-max、requests-in-flight 以及 record-retry-rate。

同时适用于两者

io-ratio 和 io-wait-ratio。

canary

如果对所有东西都进行了监控，那么 canary 就不是必需的，不过对于多层监控来说，canary 可能还是有必要的。我们可以每分钟往源集群的某个特定主题上发送一个事件，然后尝试从目标集群读取这个事件。如果这个事件在给定的时间之后才到达，那么就发出告警，说明 MirrorMaker 出现了延迟或者已经不正常了。

8.3.3 MirrorMaker调优

MirrorMaker 集群的大小取决于对吞吐量的需求和对延迟的接受程度。如果不允许有任何延迟，那么 MirrorMaker 集群的容量需要能够支撑吞吐量的上限。如果可以容忍一些延迟，那么可以在 95%~99% 的时间里只使用 75%~80% 的容量。在吞吐量高峰时可以允许一些延迟，高峰期结束时，因为 MirrorMaker 有一些空余容量，可以很容易地消除延迟。

你可能想通过消费者的线程数（通过 num.streams 参数配置）来衡量 MirrorMaker 的吞吐量。我们可以提供一些参考数据（LinkedIn 使用 8 个消费者可以达到 6MB/s 的吞吐量，使用 16 个则可以达到 12MB/s），不过实际的吞吐量取决于具体的硬件、数据中心或云服务提供商，所以需要自己进行测试。Kafka 提供了 kafka-performance-producer 工具，用于在源集群上制造负载，然后启动 MirrorMaker 对这个负载进行镜像。分别为 MirrorMaker 配置 1、2、4、8、16、24 和 32 个消费者线程，并观察性能在哪个点开始下降，然后将 num.streams 的值设置为一个小于当前点的整数。如果数据经过压缩（因为网络带宽是跨集群镜像的瓶颈，所以建议将数据压缩后再传输），那么 MirrorMaker 还要负责解压并重新压缩这些数据。这样会消耗很多的 CPU 资源，所以在增加线程数量时，要注意观察 CPU 的使用情况。通过这种方式，可以得到单个 MirrorMaker 实例的最大吞吐量。如果单个实例的吞吐量还达不到要求，可以增加更多的 MirrorMaker 实例和服务器。

另外，你可能想要分离比较敏感的主题，它们要求很低的延迟，所以其镜像必须尽可能地接近源集群和 MirrorMaker 集群。这样可以避免主题过于臃肿，或者避免出现失控的生产者拖慢数据管道。

我们能够对 MirrorMaker 进行的调优也就是这些了。不过，我们仍然有其他办法可以增加每个消费者和每个 MirrorMaker 的吞吐量。

如果 MirrorMaker 是跨数据中心运行的，可以在 Linux 上对网络进行优化。

- 增加 TCP 的缓冲区大小（`net.core.rmem_default`、`net.core.rmem_max`、`net.core.wmem_default`、`net.core.wmem_max`、`net.core.optmem_max`）。
- 启用时间窗口自动伸缩（`sysctl -w net.ipv4.tcp_window_scaling=1` 或者把 `net.ipv4.tcp_window_scaling=1` 添加到 /etc/sysctl.conf）。
- 减少 TCP 慢启动时间（将 `/proc/sys/net/ipv4/tcp_slow_start_after_idle` 设为 `0`）。

要注意，在 Linux 上进行网络调优包含了太多复杂的内容。为了了解更多参数和细节，建议阅读相关的网络调优指南。例如，由 Sandra K.Johnson 等人合著的 *Performance tuning for Linux servers*。

除此以外，你可能还想对 MirrorMaker 里的生产者和消费者进行调优。首先，你想知道生产者或消费者是不是瓶颈所在——生产者是否在等待消费者提供更多的数据，或者其他的什么？通过查看生产者和消费者的度量指标就可以知道问题所在了，如果其中的一个进程空闲，而另外一个很忙，那么就知道该对哪个进行调优了。另外一种办法是查看线程转储（thread dump），可以使用 jstack 获得线程转储。如果 MirrorMaker 的大部分时间用在轮询上，那么说明消费者出现了瓶颈，如果大部分时间用在发送上，那么就是生产者出现了瓶颈。

如果需要对生产者进行调优，可以使用下列参数。

`max.in.flight.requests.per.connection`

　　默认情况下，MirrorMaker 只允许存在一个处理中的请求。也就是说，生产者在发送下一个消息之前，当前发送的消息必须得到目标集群的确认。这样会对吞吐量造成限制，特别是当 broker 在对消息

进行确认之前出现了严重的延迟。MirrorMaker 之所以要限定请求的数量，是因为有些消息在得到成功确认之前需要进行重试，而这是唯一能够保证消息次序的方法。如果不在乎消息的次序，那么可以通过增加 `max.in.flight.requests.per.connection` 的值来提升吞吐量。

`linger.ms` 和 `batch.size`

如果在进行监控时发现生产者总是发送未填满的批次（比如，度量指标 `batch-size-avg` 和 `batch-size-max` 的值总是比 `batch.size` 低），那么就可以通过增加一些延迟来提升吞吐量。通过增加 `latency.ms` 可以让生产者在发送批次之前等待几毫秒，让批次填充更多的数据。如果发送的数据都是满批次的，同时还有空余的内存，那么可以配置更大的 `batch.size`，以便发送更大的批次。

下面的配置用于提升消费者的吞吐量。

- `range`。MirrorMaker 默认使用 `range` 策略（用于确定将哪些分区分配给哪个消费者的算法）进行分区分配。`range` 策略有一定的优势，这也就是为什么它会成为默认的策略。不过 `range` 策略会导致不公平现象。对于 MirrorMaker 来说，最好可以把策略改为 `round robin`，特别是在镜像大量的主题和分区的时候。要将策略改为 `round robin` 算法，需要在消费者配置属性文件里加上 `partition.assignment.strategy=org.apache.kafka.clie`
- `fetch.max.bytes`。如果度量指标显示 `fetch-size-avg` 和 `fetch-size-max` 的数值与 `fetch.max.bytes` 很接近，说明消费者读取的数据已经接近上限。如果有更多的可用内存，可以配置更大的 `fetch.max.bytes`，消费者就可以在每个请求里读取更多的数据。
- `fetch.min.bytes` 和 `fetch.max.wait`。如果度量指标 `fetch-rate` 的值很高，说明消费者发送的请求太多了，而且获取不到足够的数据。这个时候可以配置更大的 `fetch.min.bytes` 和 `fetch.max.wait`，这样消费者的每个请求就可以获取到更多的数据，broker 会等到有足够多的可用数据时才将响应返回。

8.4　其他跨集群镜像方案

我们深入了解了 MirrorMaker，因为 MirrorMaker 是 Kafka 的一部分。不过在实际使用当中，MirrorMaker 也存在一些不足。在 MirrorMaker 之外，还有其他的一些替代方案，它们解决了 MirrorMaker 的局限性和复杂性问题。

8.4.1 优步的uReplicator

优步在他们的 Kafka 集群上大规模地使用 MirrorMaker，不过，随着主题和分区的增加以及集群吞吐量的增长，他们开始面临一些问题。

再均衡延迟

MirrorMaker 中的消费者只是普通的消费者，在增加 MirrorMaker 的线程和实例、重启 MirrorMaker 实例或往白名单里添加新主题时，消费者都需要进行再均衡。正如在第 4 章里所看到的那样，再均衡要求关闭所有的消费者，直到新的分区被分配给消费者。如果主题和分区的数量很大，整个过程需要很长的时间。如果使用了旧版本的消费者则更是如此，比如像优步那样。有时候，这会造成 5~10 分钟的不可用，导致镜像过程延后，堆积大量的待镜像数据，需要更长的时间进行恢复，这将给其他消费者带来很大的延迟。

难以增加新主题

因为白名单使用了正则表达式进行主题匹配，每次新增主题时，MirrorMaker 都需要进行再均衡。我们已经看到优步在这方面所遭遇的痛苦。后来，为了避免意外的再均衡，他们把每一个需要镜像的主题都列了出来，这意味着他们需要手动往白名单里添加新主题，不过这样最起码可以保证再均衡只会在进行维护时发生，而不是在每次添加新主题时发生。不过不管怎样，经常性的维护是避免不了的。如果没有做好维护工作，不同的实例可能拥有不同的主题列表，MirrorMaker 就会无休止地进行再均衡，因为消费者无法就它们所订阅的主题达成一致。

为了解决上述问题，优步开发了 uReplicator。他们使用 Apache Helix（以下简称 Helix）作为中心控制器（具有高可用性），控制器管理着主题列表和分配给每个 uReplicator 实例的分区。管理员通过 REST API 添加新主题，uReplicator 负责将分区分配给不同的消费者。优步使用自己开发的 Helix Consumer 替换 MirrorMaker 里的 Kafka Consumer。Helix Consumer 接受由 Helix 控制器分配的分区，而不是在消费者间进行再均衡（更多细节参考第 4 章），从而避免了再均衡，并改为监听来自 Helix 控制器的分配变更事件。

优步在他们的博客上分享了 uReplicator 的架构细节及其所经历的改进过程。到目前为止，我们并不知道是否还有其他公司在使用 uReplicator。或许大部分公司都还达不到 Uber 那样的规模，也没有遇到相同的问题，又或者新引入的 Helix 对于他们来说需要进行额外的学习和管理，增加了整个项目的复杂性。

8.4.2　Confluent的Replicator

在优步开发 uReplicator 的同时，Confluent 也开发了他们的 Replicator。除了名字有点相似外，它们之间没有任何共同点，所要解决的问题也不一样。Replicator 为 Confluent 的企业用户解决了他们在使用 MirrorMaker 进行多集群部署时所遇到的问题。

分散的集群配置

　　MirrorMaker 只能做到源集群和目标集群之间的数据同步，而主题可以有不同的分区、不同的复制系数和不同的配置。如果将源集群的保留时间从 1 周改为 3 周，但忘记给灾备集群也做同样的修改，一旦灾备集群发生了失效备援，就会丢失几周的数据。通过手动的方式对所有配置进行同步很容易出错，而且如果系统出现了不同步，会导致下游的应用或者镜像进程失效。

在集群管理方面所面临的挑战

　　MirrorMaker 一般是以多实例的集群方式进行部署的，这意味着它本身也需要进行部署、监控和配置管理。两个配置文件和大量的配置参数让 MirrorMaker 的配置管理变得极具挑战性。如果集群超过了两个，而且集群间的复制是双向的，那么情况会更加严峻。如果有 3 个双活集群，就有 6 个 MirrorMaker 集群需要进行部署、监控和配置，而且每个集群至少需要 3 个实例。如果有 5 个双活集群，就需要 20 个 MirrorMaker 集群。

为了减轻 IT 部门的负担，Confluent 将 Replicator 实现为 Connect 的源连接器，从 Kafka 集群读取数据，而不是从数据库读取。在第 7 章介绍 Connect 的架构时，我们知道，连接器会将工作分配给多个任务。在 Replicator 里，每个任务包含了一个消费者和一个生产者。Connect 根据实际情况将不同的任务分配给不同的 worker 节点，因此单个服务器上可能会有多个任务，或者任务被分散在多个服务器上，这样就避免了手动去配置每个 MirrorMaker 实例需要多少个线程以及每台服务器需要多少个 MirrorMaker 实例。Connect 还提供了 REST API，用于集中管理连接器和任务。假设大部分 Kafka 都部署了

Connect（比如为了将数据库的变更事件写入 Kafka），那么通过在 Connect 里运行 Replicator，就可以减少需要管理的集群数量。另一个重大的改进在于，Replicator 不仅会从 Kafka 主题复制数据，它还会从 Zookeeper 上复制主题的配置信息。

8.5　总结

本章从解释为什么需要多个 Kafka 集群开始，介绍了几种从简单到复杂的多集群架构，还介绍了 Kafka 失效备援的实现细节，并比较了当前几种可用的方案。接下来介绍了一些可用的工具，从 MirrorMaker 开始，说明了在生产环境中使用 MirrorMaker 要注意的细节问题，最后介绍了 MirrorMaker 之外的两个替代方案，用于弥补 MirrorMaker 本身的不足。

不管最终选择哪一种架构和工具，对多集群配置和镜像管道进行测试总是少不了的。因为 Kafka 多集群管理比关系型数据库要简单得多，所以很多组织总是忽视了对它进行适当的设计、规划、测试、自动化部署、监控和维护。重视多集群的管理问题，并把它作为组织的全盘灾备计划或多区域计划的一部分，才有可能更好地管理好多个 Kafka 集群。

第 9 章　管理 Kafka

Kafka 提供了一些命令行工具，用于管理集群的变更。这些工具使用 Java 类实现，Kafka 提供了一些脚本来调用这些 Java 类。不过，它们只提供了一些基本的功能，无法完成那些复杂的操作。本章将介绍一些工具，它们是 Kafka 开放源码项目的一部分。Kafka 社区也开发了很多高级的工具，我们可以在 Apache Kafka 网站上找到它们，不过它们并不属于 Kafka 项目。

> 管理操作授权
>
> 虽然 Kafka 实现了操作主题的认证和授权控制，但还不支持集群的其他大部分操作。也就是说，在没有认证的情况下也可以使用这些命令行工具，在没有安全检查和审计的情况下也可以执行诸如主题变更之类的操作。不过这些功能正在开发当中，应该很快就能

发布。

9.1 主题操作

使用 kafka-topics.sh 工具可以执行主题的大部分操作（配置变更部分已经被弃用并被移动到 kafka-configs.sh 工具当中）。我们可以用它创建、修改、删除和查看集群里的主题。要使用该工具的全部功能，需要通过 `--zookeeper` 参数提供 Zookeeper 的连接字符串。在下面的例子里，Zookeeper 的连接字符串是 `zoo1.example.com:2181/kafka-cluster`。

> 检查版本
>
> Kafka 的大部分命令行工具直接操作 Zookeeper 上的元数据，并不会连接到 broker 上。因此，要确保所使用工具的版本与集群里的 broker 版本相匹配。直接使用集群 broker 自带的工具是最保险的。

9.1.1 创建主题

在集群里创建一个主题需要用到 3 个参数。这些参数是必须提供的，尽管有些已经有了 broker 级别的默认值。

主题名字

  想要创建的主题的名字。

复制系数

  主题的副本数量。

分区

主题的分区数量。

> 指定主题配置
>
> 可以在创建主题时显式地指定复制系数或者对配置进行覆盖，不过我们不打算在这里介绍如何做到这些。稍后会介绍如何进行配置覆盖，它们是通过向 kafka-topics.sh 传递 `--config` 参数来实现的。本章还会

介绍分区的重分配。

主题名字可以包含字母、数字、下划线以及英文状态下的破折号和句号。

> 主题的命名
>
> 主题名字的开头部分包含两个下划线是合法的，但不建议这么做。具有这种格式的主题一般是集群的内部主题（比如 `__consumer_offsets` 主题用于保存消费者群组的偏移量）。也不建议在单个集群里使用英文状态下的句号和下划线来命名，因为主题的名字会被用在度量指标上，句号会被替换成下划线（比如“topic.1”会变成“topic_1”）。

试着运行下面的命令：

```
kafka-topics.sh --zookeeper <zookeeper connect> --create --topic <string>
--replication-factor <integer> --partitions <integer>
```

这个命令将会创建一个主题，主题的名字为指定的值，并包含了指定数量的分区。集群会为每个分区创建指定数量的副本。如果为集群指定了基于机架信息的副本分配策略，那么分区的副本会分布在不同的机架上。如果不需要基于机架信息的分配策略，可以指定参数 `--disable-rack-aware`。

示例：使用以下命令创建一个叫作 my-topic 的主题，主题包含 8 个分区，每个分区拥有两个副本。

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --create
--topic my-topic --replication-factor 2 --partitions 8
Created topic "my-topic".
#
```

> 忽略重复创建主题的错误
>
> 在自动化系统里调用这个脚本时，可以使用 `--if-not-exists` 参数，这样即使主题已经存在，也不会抛出重复创建主题的错误。

9.1.2 增加分区

有时候，我们需要为主题增加分区数量。主题基于分区进行伸缩和复制，增加分区主要是为了扩展主题容量或者降低单个分区的吞吐量。如果要在单个消费者群组内运行更多的消费者，那么主题数量也需要相应增加，因为一个分区只能由群组里的一个消费者读取。

> 调整基于键的主题
>
> 从消费者角度来看，为基于键的主题添加分区是很困难的。因为如果改变了分区的数量，键到分区之间的映射也会发生变化。所以，对于基于键的主题来说，建议在一开始就设置好分区数量，避免以后对其进行调整。

> 忽略主题不存在的错误
>
> 在使用 `--alter` 命令修改主题时，如果指定了 `--if-exists` 参数，主题不存在的错误就会被忽略。如果要修改的主题不存在，该命令并不会返回任何错误。在主题不存在的时候本应该创建主题，但它却把错误隐藏起来，因此不建议使用这个参数。

示例：将 my-topic 主题的分区数量增加到 16。

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--alter -- topic my-topic --partitions 16
WARNING: If partitions are increased for a topic that has a key,
the partition logic or ordering of the messages will be affected
Adding partitions succeeded!
#
```

> 减少分区数量
>
> 我们无法减少主题的分区数量。因为如果删除了分区，分区里的数据也一并被删除，导致数据不一致。我们也无法将这些数据分配给其他分区，因为这样做很难，而且会出现消息乱序。所以，如果一定要减少分区数量，只能删除整个主题，然后重新创建它。

9.1.3 删除主题

如果一个主题不再被使用，只要它还存在于集群里，就会占用一定数量的磁盘空间和文件句柄。把它删除就可以释放被占用的资源。为了能够删除主题，broker 的 `delete.topic.enable` 参数必须被设置为 `true`。如果该参数被设为 `false`，删除主题的请求会被忽略。

 删除主题会丢弃主题里的所有数据。这是一个不可逆的操作，所以在执行时要十分小心。

示例：删除 my-topic 主题。

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--delete -- topic my-topic
Topic my-topic is marked for deletion.
Note: This will have no impact if delete.topic.enable is not set
to true.
#
```

9.1.4 列出集群里的所有主题

可以使用主题工具列出集群里的所有主题。每个主题占用一行输出，主题之间没有特定的顺序。

示例：列出集群里的所有主题。

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--list
my-topic - marked for deletion
other-topic
#
```

9.1.5 列出主题详细信息

主题工具还能用来获取主题的详细信息。信息里包含了分区数量、主题的覆盖配置以及每个分区的副本清单。如果通过 `--topic` 参数指定特定的主题，就可以只列出指定主题的详细信息。

示例：列出集群里所有主题的详细信息。

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --descri
Topic:other-topic       PartitionCount:8        ReplicationFactor:2  Confi
Topic:other-topic       Partition: 0      ...   Replicas: 1,0        Isr:
Topic:other-topic       Partition: 1      ...   Replicas: 0,1        Isr:
Topic:other-topic       Partition: 2      ...   Replicas: 1,0        Isr:
Topic:other-topic       Partition: 3      ...   Replicas: 0,1        Isr:
Topic:other-topic       Partition: 4      ...   Replicas: 1,0        Isr:
Topic:other-topic       Partition: 5      ...   Replicas: 0,1        Isr:
Topic:other-topic       Partition: 6      ...   Replicas: 1,0        Isr:
Topic:other-topic       Partition: 7      ...   Replicas: 0,1        Isr:
#
```

describe 命令还提供了一些参数，用于过滤输出结果，这在诊断集群问题时会很有用。不要为这些参数指定 --topic 参数（因为这些参数的目的是为了找出集群里所有满足条件的主题和分区）。这些参数也无法与 list 命令一起使用（最后一部分会详细说明原因）。

使用 --topics-with-overrides 参数可以找出所有包含覆盖配置的主题，它只会列出包含了与集群不一样配置的主题。

有两个参数可用于找出有问题的分区。使用 --under-replicated-partitions 参数可以列出所有包含不同步副本的分区。使用 --unavailable-partitions 参数可以列出所有没有首领的分区，这些分区已经处于离线状态，对于生产者和消费者来说是不可用的。

示例：列出包含不同步副本的分区。

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--describe --under-replicated-partitions
        Topic: other-topic      Partition: 2    Leader: 0    Replicas: 1,0
        Isr: 0
        Topic: other-topic      Partition: 4    Leader: 0    Replicas: 1,0
        Isr: 0
#
```

9.2 消费者群组

在 Kafka 里，有两个地方保存着消费者群组的信息。对于旧版本的消费者来说，它们的信息保存在 Zookeeper 上；对于新版本的消费者来说，它们的信息保存在 broker 上。kafka-consumer-groups.sh 工具可以用于列出上述两种消费者群组。它也可以用于删除消费者群组和偏

移量信息，不过这个功能仅限于旧版本的消费者群组（信息保存在 Zookeeper 上）。在对旧版本的消费者群组进行操作时，需要通过 `--zookeeper` 参数指定 Zookeeper 的地址；在对新版本的消费者群组进行操作时，则需要使用 `--bootstrap-server` 参数指定 broker 的主机名和端口。

9.2.1　列出并描述群组

在使用旧版本的消费者客户端时，可以使用 `--zookeeper` 和 `--list` 参数列出消费者群组；在使用新版本的消费者客户端时，则要使用 `--bootstrap-server`、`--list` 和 `--new-consumer` 参数。

示例：列出旧版本的消费者群组。

```
# kafka-consumer-groups.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --list
console-consumer-79697
myconsumer
#
```

示例：列出新版本的消费者群组。

```
# kafka-consumer-groups.sh --new-consumer --bootstrap-server
kafka1.example.com:9092/kafka-cluster --list
kafka-python-test
my-new-consumer
#
```

对于列出的任意群组来说，使用 `--describe` 代替 `--list`，并通过 `--group` 指定特定的群组，就可以获取该群组的详细信息。它会列出群组里所有主题的信息和每个分区的偏移量。

示例：获取旧版本消费者群组 testgroup 的详细信息。

```
# kafka-consumer-groups.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--describe --group testgroup
GROUP                          TOPIC                          PARTITION
CURRENT-OFFSET LOG-END-OFFSET  LAG                 OWNER
myconsumer                     my-topic                       0
1688           1688             0
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer                     my-topic                       1
```

```
1418            1418               0
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer                        my-topic                      2
1314            1315               1
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer                        my-topic                      3
2012            2012               0
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer                        my-topic                      4
1089            1089               0
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer                        my-topic                      5
1429            1432               3
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer                        my-topic                      6
1634            1634               0
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
myconsumer                        my-topic                      7
2261            2261               0
myconsumer_host1.example.com-1478188622741-7dab5ca7-0
#
```

输出结果里包含了如表 9-1 所示的字段。

表**9-1**：输出结果中的字段

字段	描述
GROUP	消费者群组的名字
TOPIC	正在被读取的主题名字
PARTITION	正在被读取的分区 ID
CURRENT-OFFSET	消费者群组最近提交的偏移量，也就是消费者在分区里读取的当前位置
LOG-END-OFFSET	当前高水位偏移量，也就是最近一个被读取消息的偏移量，同时也是最近一个被提交到集群的偏移量
LAG	消费者的 CURRENT-OFFSET 和 broker 的 LOG-END-OFFSET 之间的差距
OWNER	消费者群组里正在读取该分区的消费者。这是一个消费者的 ID，不一定包含消费者的主机名

9.2.2 删除群组

只有旧版本的消费者客户端才支持删除群组的操作。删除群组操作将从 Zookeeper 上移除整个群组，包括所有已保存的偏移量。在执行该操作之前，必须关闭所有的消费者。如果不先执行这一步，可能会导致消费者出现不可预测的行为，因为群组的元数据已经从 Zookeeper 上移除了。

示例：删除消费者群组 testgroup。

```
# kafka-consumer-groups.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --delete --group testgroup
Deleted all consumer group information for group testgroup in
zookeeper.
#
```

该命令也可以用于在不删除整个群组的情况下删除单个主题的偏移量。再次强调，在进行删除操作之前，需要先关闭消费者，或者不要让它们读取即将被删除的主题。

示例：从消费者群组 testgroup 里删除 my-topic 主题的偏移量。

```
# kafka-consumer-groups.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --delete --group testgroup
--topic my-topic
Deleted consumer group information for group testgroup topic
my-topic in zoo keeper.
#
```

9.2.3 偏移量管理

除了可以显示和删除消费者群组（使用了旧版本消费者客户端）的偏移量外，还可以获取偏移量，并保存批次的最新偏移量，从而实现偏移量的重置。在需要重新读取消息或者因消费者无法正常处理消息（比如包含了非法格式的消息）需要跳过偏移量时，需要进行偏移量重置。

> 管理已经提交到 **Kafka** 的偏移量
>
> 目前，还没有工具可以用于管理由消费者客户端提交到 Kafka 的偏移量，管理功能只对提交到 Zookeeper 的偏移量可用。另外，为了能够管理提交到 Kafka 的消费者群组偏移量，需要在客户端使用相应的 API 来提交群组的偏移量。

01. 导出偏移量

Kafka 没有为导出偏移量提供现成的脚本，不过可以使用 kafka-run-class.sh 脚本调用底层的 Java 类来实现导出。在导出偏移量时，会生成一个文件，文件里包含了分区和偏移量的信息。偏移量信息以一种导入工具能够识别的格式保存在文件

里。每个分区在文件里占用一行，格式为：/consumers/GROUPNAME/offsets/topic/TOPICNAME/PARTITIONID-0:OFFSET。

示例：将群组 testgroup 的偏移量导出到 offsets 文件里。

```
# kafka-run-class.sh kafka.tools.ExportZkOffsets
--zkconnect zoo1.example.com:2181/kafka-cluster --group testgroup
--output-file offsets
# cat offsets
/consumers/testgroup/offsets/my-topic/0:8905
/consumers/testgroup/offsets/my-topic/1:8915
/consumers/testgroup/offsets/my-topic/2:9845
/consumers/testgroup/offsets/my-topic/3:8072
/consumers/testgroup/offsets/my-topic/4:8008
/consumers/testgroup/offsets/my-topic/5:8319
/consumers/testgroup/offsets/my-topic/6:8102
/consumers/testgroup/offsets/my-topic/7:12739
#
```

02. 导入偏移量

偏移量导入工具与导出工具做的事情刚好相反，它使用之前导出的文件来重置消费者群组的偏移量。一般情况下，我们会导出消费者群组的当前偏移量，并将导出的文件复制一份（这样就有了一个备份），然后修改复制文件里的偏移量。这里要注意，在使用导入命令时，不需要使用 `--group` 参数，因为文件里已经包含了消费者群组的名字。

> 先关闭消费者
>
> 在导入偏移量之前，必须先关闭所有的消费者。如果消费者群组处于活跃状态，它们不会读取新的偏移量，反而有可能将导入的偏移量覆盖掉。

示例：从 offsets 文件里将偏移量导入到消费者群组 testgroup。

```
# kafka-run-class.sh kafka.tools.ImportZkOffsets --zkconnect
zoo1.example.com:2181/kafka-cluster --input-file offsets
#
```

9.3 动态配置变更

我们可以在集群处于运行状态时覆盖主题配置和客户端的配额参数。我们打算在未来增加更多的动态配置参数，这也是为什么这些参数被单独放进了 kafka-configs.sh。这样就可以为特定的主题和客户端指定配置参数。一旦设置完毕，它们就成为集群的永久配置，被保存在 Zookeeper 上，broker 在启动时会读取它们。不管是在工具里还是文档里，它们所说的动态配置参数都是基于“主题”实例或者“客户端”实例的，都是可以被“覆盖”的。

与之前介绍的工具一样，这里也需要通过 `--zookeeper` 参数提供 Zookeeper 集群的连接字符串。在下面的例子里，Zookeeper 的连接字符串是“zoo1.example.com:2181/kafka-cluster”。

9.3.1 覆盖主题的默认配置

为了满足不同的使用场景，主题的很多参数都可以进行单独的设置。它们大部分都有 broker 级别的默认值，在没有被覆盖的情况下使用默认值。

更改主题配置的命令格式如下。

```
kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--alter --entity-type topics --entity-name <topic name>
--add-config <key>=<value>[,<key>=<value>...]
```

可用的主题配置参数（键）如表 9-2 所示。

表**9-2**：可用的主题配置参数

配置项	描述
cleanup.policy	如果被设置为 `compact`，只有最新包含了指定 key 的消息会被保留下来（压缩日志），其他的被丢弃掉
compression.type	broker 在将消息批次写入磁盘时所使用的压缩类型，目前支持“gzip”、“snappy”和“lz4”
delete.retention.ms	被标识为待删除的数据能够保留多久，以 ms 为单位。该参数只对压缩日志类型的主题有效
file.delete.delay.ms	从磁盘上删除日志片段和索引之前可以等待多长时间，以 ms 为单位
flush.messages	需要收到多少个消息才能将它们刷新到磁盘
flush.ms	在将消息刷新到磁盘之前可以等待多长时间，以 ms 为单位
index.interval.bytes	日志片段的两个索引之间能够容纳的消息字节数
max.message.bytes	最大消息字节数
message.format.version	broker 将消息写入磁盘时所使用的消息格式，必须是有效的 API 版本号（比如“0.10.0”）
message.timestamp.difference.max.ms	消息自带的时间戳和 broker 收到消息时的时间戳之间最大的差值，以 ms 为单位。该参数

	只在 messsage.timestamp.type 被设为 Create-Time 时有效
message.timestamp.type	在将消息写入磁盘时使用哪一种时间戳。目前支持两种值，其中 CreateTime 指客户端指定的时间戳，而 LogAppendTime 指消息被写入分区时的时间戳
min.cleanable.dirty.ratio	日志压缩器压缩分区的频率，使用未压缩日志片段数与总日志分段数之间的比例来表示。该参数只对压缩日志类型的主题有效
min.insync.replicas	可用分区的最少同步副本
preallocate	如果被设为 true，需要为新的日志片段预分配空间
retention.bytes	主题能够保留的消息量，以字节为单位
retention.ms	主题需要保留消息多长时间，以 ms 为单位
segment.bytes	日志片段的消息字节数
segment.index.bytes	单个日志片段的最大索引字节数
segment.jitter.ms	滚动日志片断时，在 segment.ms 基础上增加的随机毫秒数
segment.ms	日志片段多长时间滚动一次，以 ms 为单位
unclean.leader.election.enable	如果被设为 true，不彻底的首领选择无效

示例：将主题 my-topic 的消息保留时间设为 1 个小时（3 600 000ms）。

```
# kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--alter -- entity-type topics --entity-name my-topic --add-config
retention.ms=3600000
Updated config for topic: "my-topic".
#
```

9.3.2　覆盖客户端的默认配置

对于 Kafka 客户端来说，只能覆盖生产者配额和消费者配额参数。这两个配额都以字节每秒为单位，表示客户端在每个 broker 上的生产速率或消费速率。也就是说，如果集群里有 5 个 broker，生产者的配额是 10MB/s，那么它可以以 10MB/s 的速率在单个 broker 上生成数据，总共的速率可以达到 50MB/s。

客户端 **ID** 与消费者群组

客户端 ID 可以与消费者群组的名字不一样。消费者可以有自己的 ID，因此不同群组里的消费者可能具有相同的 ID。在为消费者客户端设置 ID 时，最好使用能够表明它们所属群组的标识符，这样便于群组共享配额，从日志里查找负责请求的群组也更容易一些。

更改客户端配置的命令格式如下：

```
kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--alter -- entity-type clients --entity-name <client ID>
--add-config <key>=<value>[,<key>=<value>...]
```

可用的客户端配置参数（键）如表 9-3 所示。

表**9-3**：可用的客户端配置参数

配置项	描述
producer_bytes_rate	单个生产者每秒钟可以往单个 broker 上生成的消息字节数
consumer_bytes_rate	单个消费者每秒钟可以从单个 broker 读取的消息字节数

9.3.3　列出被覆盖的配置

使用命令行工具可以列出所有被覆盖的配置，从而用于检查主题或客户端的配置。与其他工具类似，这个功能通过 `--describe` 命令来实现。

示例：列出主题 my-topic 所有被覆盖的配置。

```
# kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--describe -- entity-type topics --entity-name my-topic
Configs for topics:my-topic are
retention.ms=3600000,segment.ms=3600000
#
```

只能显示主题的覆盖配置

这个命令只能用于显示被覆盖的配置，不包含集群的默认配置。目前还无法通过 Zookeeper 或 Kafka 实现动态地获取 broker 本身的配置。也就是说，在进行自动化时，如果要使用这个工具来获得主题和客户端的配置信息，必须同时为它提供集群的默认配置信息。

9.3.4　移除被覆盖的配置

动态的配置完全可以被移除，从而恢复到集群的默认配置。可以使用 `--alter` 命令和 `--delete-config` 参数来删除被覆盖的配置。

示例：删除主题 my-topic 的 `retention.ms` 覆盖配置。

```
# kafka-configs.sh --zookeeper zoo1.example.com:2181/kafka-cluster
--alter -- entity-type topics --entity-name my-topic
--delete-config retention.ms
Updated config for topic: "my-topic".
#
```

9.4 分区管理

Kafka 工具提供了两个脚本用于管理分区，一个用于重新选举首领，另一个用于将分区分配给 broker。结合使用这两个工具，就可以实现集群流量的负载均衡。

9.4.1 首选的首领选举

第 6 章提到，使用多个分区副本可以提升可靠性。不过，只有其中的一个副本可以成为分区首领，而且只有首领所在的 broker 可以进行生产和消费活动。Kafka 将副本清单里的第一个同步副本选为首领，但在关闭并重启 broker 之后，并不会自动恢复原先首领的身份。

 自动首领再均衡

> broker 有一个配置可以用于启用自动首领再均衡，不过到目前为止，并不建议在生产环境使用该功能。自动均衡会带来严重的性能问题，在大型的集群里，它会造成客户端流量的长时间停顿。

通过触发首选的副本选举，可以让 broker 重新获得首领。当该事件被触发时，集群控制器会为分区重新选择理想的首领。选举过程一般不会造成负面的影响，因为客户端可以自动跟踪首领的变化。也可以通过 kafka-preferred-replica-election.sh 工具手动触发选举。

示例：在一个包含了 1 个主题和 8 个分区的集群里启动首选的副本选举。

```
# kafka-preferred-replica-election.sh --zookeeper
zoo1.example.com:2181/kafka-cluster
Successfully started preferred replica election for partitions
Set([my-topic,5], [my-topic,0], [my-topic,7], [my-topic,4],
[my-topic,6], [my-topic,2], [my-topic,3], [my-topic,1])
#
```

因为集群包含了大量的分区，首选的副本选举有可能无法正常进行。在进行选举时，集群的元数据必须被写到 Zookeeper 的节点上，如果元数据超过了节点允许的大小（默认是 1MB），那么选举就会失败。这个时候，需要将分区清单的信息写到一个 JSON 文件里，并将请求分为多个步骤进行。JSON 文件的格式如下：

```
{
    "partitions": [
        {
            "partition": 1,
            "topic": "foo"
        },
        {
            "partition": 2,
            "topic": "foobar"
        }
    ]
}
```

示例：通过在 partitions.json 文件里指定分区清单来启动副本选举。

```
# kafka-preferred-replica-election.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --path-to-json-file
partitions.json
Successfully started preferred replica election for partitions
Set([my-topic,1], [my-topic,2], [my-topic,3])
#
```

9.4.2　修改分区副本

在某些时候，可能需要修改分区的副本。以下是一些需要修改分区副本的场景。

- 主题分区在整个集群里的不均衡分布造成了集群负载的不均衡。
- broker 离线造成分区不同步。
- 新加入的 broker 需要从集群里获得负载。

可以使用 kafka-reassign-partitions.sh 工具来修改分区。使用该工具需要经过两个步骤：第一步，根据 broker 清单和主题清单生成一组迁移步骤；第二步，执行这些迁移步骤。第三个步骤是可选的，也就

是可以使用生成的迁移步骤验证分区重分配的进度和完成情况。

为了生成迁移步骤，需要先创建一个包含了主题清单的 JSON 文件，文件格式如下（目前的版本号都是 1）：

```
{
    "topics": [
        {
            "topic": "foo"
        },
        {
            "topic": "foo1"
        }
    ],
    "version": 1
}
```

示例：为 topics.json 文件里的主题生成迁移步骤，以便将这些主题迁移到 broker 0 和 broker 1 上。

```
# kafka-reassign-partitions.sh --zookeeper
zoo1.example.com:2181/kafka-cluster
--generate --topics-to-move-json-file topics.json --broker-list 0,1
Current partition replica assignment

{"version":1,"partitions":[{"topic":"my-topic","partition":5,"replicas":[0
{"topic":"my-topic","partition":10,"replicas":[1,0]},{"topic":"mytopic","
partition":1,"replicas":[0,1]},{"topic":"my-topic","partition":4,"repli
cas":[1,0]},{"topic":"my-topic","partition":7,"replicas":[0,1]},{"topic":"
partition":6,"replicas":[1,0]},{"topic":"my-topic","partition":
3,"replicas":[0,1]},{"topic":"my-topic","partition":15,"replicas":[0,1]},
{"topic":"my-topic","partition":0,"replicas":[1,0]},{"topic":"mytopic","
partition":11,"replicas":[0,1]},{"topic":"my-topic","partition":8,"repli
cas":[1,0]},{"topic":"my-topic","partition":12,"replicas":[1,0]},{"topic":
partition":2,"replicas":[1,0]},{"topic":"my-topic","partition":
13,"replicas":[0,1]},{"topic":"my-topic","partition":14,"replicas":[1,0]},
{"topic":"my-topic", "partition":9,"replicas":[0,1]}]}
Proposed partition reassignment configuration

{"version":1,"partitions":[{"topic":"my-topic","partition":5,"replicas":[0
{"topic":"my-topic","partition":10,"replicas":[1,0]},{"topic":"mytopic","
partition":1,"replicas":[0,1]},{"topic":"my-topic","partition":4,"repli
cas":[1,0]},{"topic":"my-topic","partition":7,"replicas":[0,1]},{"topic":"
partition":6,"replicas":[1,0]},{"topic":"my-topic", "partition":
15,"replicas":[0,1]},{"topic":"my-topic","partition":0,"replicas":[1,0]},
{"topic":"my-topic","partition":3,"replicas":[0,1]},{"topic":"mytopic","
partition":11,"replicas":[0,1]},{"topic":"my-topic","partition":8,"repli
cas":[1,0]},{"topic":"my-topic","partition":12,"replicas":[1,0]},{"topic":
partition":13,"replicas":[0,1]},{"topic":"my-topic","partition":
```

```
2,"replicas":[1,0]},{"topic":"my-topic","partition":14,"replicas":[1,0]},
{"topic":"my-topic","partition":9,"replicas":[0,1]}]}
#
```

broker 的 ID 以逗号分隔，并作为参数提供给命令行工具。这个工具会在标准控制台上输出两个 JSON 对象，分别描述了当前的分区分配情况以及建议的分区分配方案。这些 JSON 对象的格式如下：`{"partitions": [{"topic": "my-topic", "partition": 0, "replicas": [1,2] }], "version":1}`。

可以把第一个 JSON 对象保存起来，以便在必要的时候进行回滚。第二个 JSON 对象应该被保存到另一个文件里，作为 kafka-reassign-partitions.sh 工具的输入来执行第二个步骤。

示例：使用 reassign.json 来执行建议的分区分配方案。

```
# kafka-reassign-partitions.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --execute
--reassignment-json-file reassign.json
Current partition replica assignment

{"version":1,"partitions":[{"topic":"my-topic","partition":5,"replicas":[0
{"topic":"my-topic","partition":10,"replicas":[1,0]},{"topic":"mytopic","
partition":1,"replicas":[0,1]},{"topic":"my-topic","partition":4,"repli
cas":[1,0]},{"topic":"my-topic","partition":7,"replicas":[0,1]},{"topic":"
partition":6,"replicas":[1,0]},{"topic":"my-topic", "partition":
3,"replicas":[0,1]},{"topic":"my-topic","partition":15,"replicas":[0,1]},
{"topic":"my-topic","partition":0,"replicas":[1,0]},{"topic":"mytopic","
partition":11,"replicas":[0,1]},{"topic":"my-topic","partition":8,"repli
cas":[1,0]},{"topic":"my-topic","partition":12,"replicas":[1,0]},{"topic":
partition":2,"replicas":[1,0]},{"topic":"my-topic", "partition":
13,"replicas":[0,1]},{"topic":"my-topic","partition":14,"replicas":[1,0]},
{"topic":"my-topic","partition":9,"replicas":[0,1]}]}

Save this to use as the --reassignment-json-file option during
rollback
Successfully started reassignment of partitions {"version":1,"partitions":
[{"topic":"my-topic","partition":5,"replicas":[0,1]},{"topic":"mytopic","
partition":0,"replicas":[1,0]},{"topic":"my-topic","partition":7,"repli
cas":[0,1]},{"topic":"my-topic","partition":13,"replicas":[0,1]},{"topic":
partition":4,"replicas":[1,0]},{"topic":"my-topic", "partition":
12,"replicas":[1,0]},{"topic":"my-topic","partition":6,"replicas":[1,0]},
{"topic":"my-topic","partition":11,"replicas":[0,1]},{"topic":"mytopic","
partition":10,"replicas":[1,0]},{"topic":"my-topic","partition":9,"repli
cas":[0,1]},{"topic":"my-topic","partition":2,"replicas":[1,0]},{"topic":"
partition":14,"replicas":[1,0]},{"topic":"my-topic","partition":
3,"replicas":[0,1]},{"topic":"my-topic","partition":1,"replicas":[0,1]},
{"topic":"my-topic","partition":15,"replicas":[0,1]},{"topic":"my-topic",
```

```
"partition":8,"replicas":[1,0]}]}
#
```

该命令会将指定分区的副本重新分配到新的 broker 上。集群控制器通过为每个分区添加新副本实现重新分配（增加复制系数）。新的副本将从分区的首领那里复制所有数据。根据分区大小的不同，复制过程可能需要花一些时间，因为数据是通过网络复制到新副本上的。在复制完成之后，控制器将旧副本从副本清单里移除（恢复到原先的复制系数）。

为重新分配副本进行网络优化

如果要从单个 broker 上移除多个分区，比如将 broker 移出集群，那么在重新分配副本之前最好先关闭或者重启 broker。这样，这个 broker 就不再是任何一个分区的首领，它的分区就可以被分配给集群里的其他 broker（只要没有启用自动首领选举）。这可以显著提升重分配的性能，并减少对集群的影响，因为复制流量将会被分发给多个 broker。

在重分配进行过程中或者完成之后，可以使用 kafka-reassign-partitions.sh 工具验证重分配的状态。它可以显示重分配的进度、已经完成重分配的分区以及错误信息（如果有的话）。为了做到这一点，需要在执行过程中使用 JSON 对象文件。

示例：验证 reassign.json 文件里指定的分区重分配情况。

```
# kafka-reassign-partitions.sh --zookeeper
zoo1.example.com:2181/kafka-cluster --verify
--reassignment-json-file reassign.json
Status of partition reassignment:
Reassignment of partition [my-topic,5] completed successfully
Reassignment of partition [my-topic,0] completed successfully
Reassignment of partition [my-topic,7] completed successfully
Reassignment of partition [my-topic,13] completed successfully
Reassignment of partition [my-topic,4] completed successfully
Reassignment of partition [my-topic,12] completed successfully
Reassignment of partition [my-topic,6] completed successfully
Reassignment of partition [my-topic,11] completed successfully
Reassignment of partition [my-topic,10] completed successfully
Reassignment of partition [my-topic,9] completed successfully
Reassignment of partition [my-topic,2] completed successfully
Reassignment of partition [my-topic,14] completed successfully
Reassignment of partition [my-topic,3] completed successfully
Reassignment of partition [my-topic,1] completed successfully
```

```
Reassignment of partition [my-topic,15] completed successfully
Reassignment of partition [my-topic,8] completed successfully
#
```

分批重分配

分区重分配对集群的性能有很大影响，因为它会引起内存页缓存发生变化，并占用额外的网络和磁盘资源。将重分配过程拆分成多个小步骤可以将这种影响降到最低。

9.4.3 修改复制系数

分区重分配工具提供了一些特性，用于改变分区的复制系数，这些特性并没有在文档里说明。如果在创建分区时指定了错误的复制系数（比如在创建主题时没有足够多可用的 broker），那么就有必要修改它们。这可以通过创建一个 JSON 对象来完成，该对象使用分区重新分配的执行步骤中使用的格式，显式指定分区所需的副本数量。集群将完成重分配过程，并使用新的复制系数。

例如，假设主题 my-topic 有一个分区，该分区的复制系数为 1。

```
{
    "partitions": [
        {
            "topic": "my-topic",
            "partition": 0,
            "replicas": [
                1
            ]
        }
    ],
    "version": 1
}
```

在分区重新分配的执行步骤中使用以下 JSON 可以将复制系数改为 2。

```
{
    "partitions": [
        {
            "partition": 0,
            "replicas": [
                1,
```

```
                2
            ],
            "topic": "my-topic"
        }
    ],
    "version": 1
}
```

也可以通过类似的方式减小分区的复制系数。

9.4.4　转储日志片段

如果需要查看某个特定消息的内容，比如一个消费者无法处理的“毒药”消息，可以使用工具来解码分区的日志片段。该工具可以让你在不读取消息的情况下查看消息的内容。它接受一个以逗号分隔的日志片段文件清单作为参数，并打印出每个消息的概要信息和数据内容。

示例：解码日志片段 00000000000052368601.log，显示消息的概要信息。

```
# kafka-run-class.sh kafka.tools.DumpLogSegments --files
00000000000052368601.log
Dumping 00000000000052368601.log
Starting offset: 52368601
offset: 52368601 position: 0 NoTimestampType: -1 isvalid:true
payloadsize: 661 magic: 0 compresscodec: GZIPCompressionCodec crc:
1194341321
offset: 52368603 position: 687 NoTimestampType: -1 isvalid: true
payloadsize:895  magic: 0 compresscodec: GZIPCompressionCodec crc:
278946641
offset: 52368604 position: 1608 NoTimestampType: -1 isvalid: true
payloadsize:665  magic: 0 compresscodec: GZIPCompressionCodec crc:
3767466431
offset: 52368606 position: 2299 NoTimestampType: -1 isvalid: true
payloadsize:932  magic: 0 compresscodec: GZIPCompressionCodec crc:
2444301359
...
```

示例：解码日志片段 00000000000052368601.log，显示消息的数据内容。

```
# kafka-run-class.sh kafka.tools.DumpLogSegments --files
00000000000052368601.log --print-data-log
offset: 52368601 position: 0 NoTimestampType: -1 isvalid: true
payloadsize: 661 magic: 0 compresscodec: GZIPCompressionCodec crc:
1194341321 payload: test message 1
```

```
offset: 52368603 position: 687 NoTimestampType: -1 isvalid: true
payloadsize:895  magic: 0 compresscodec: GZIPCompressionCodec crc:
278946641 payload: test message 2
offset: 52368604 position: 1608 NoTimestampType: -1 isvalid: true
payloadsize:665  magic: 0 compresscodec: GZIPCompressionCodec crc:
3767466431 payload: test message 3
offset: 52368606 position: 2299 NoTimestampType: -1 isvalid: true
payloadsize:932  magic: 0 compresscodec: GZIPCompressionCodec crc:
2444301359 payload: test message 4
...
```

这个工具也可以用于验证日志片段的索引文件。索引用于在日志片段里查找消息，如果索引文件损坏，会导致消费者在读取消息时出现错误。broker 在不正常启动（比如之前没有正常关闭）时会自动执行这个验证过程，不过也可以手动执行它。有两个参数可以用于指定不同程度的验证，`--index-sanity-check` 将会检查无用的索引，而 `--verify-index-only` 将会检查索引的匹配度，但不会打印出所有的索引。

示例：验证日志片段 00000000000052368601.log 索引文件的正确性。

```
# kafka-run-class.sh kafka.tools.DumpLogSegments --files
00000000000052368601.index,00000000000052368601.log
--index-sanity-check
Dumping 00000000000052368601.index
00000000000052368601.index passed sanity check.
Dumping 00000000000052368601.log
Starting offset: 52368601
offset: 52368601 position: 0 NoTimestampType: -1 isvalid: true
payloadsize: 661 magic: 0 compresscodec: GZIPCompressionCodec crc:
1194341321
offset: 52368603 position: 687 NoTimestampType: -1 isvalid: true
payloadsize:895  magic: 0 compresscodec: GZIPCompressionCodec crc:
278946641
offset: 52368604 position: 1608 NoTimestampType: -1 isvalid: true
payloadsize:665  magic: 0 compresscodec: GZIPCompressionCodec crc:
3767466431
...
```

9.4.5　副本验证

分区复制的工作原理与消费者客户端类似：跟随者 broker 定期将上一个偏移量到当前偏移量之间的数据复制到磁盘上。如果复制停止并重启，它会从上一个检查点继续复制。如果之前复制的日志片段被删

除，跟随者不会做任何补偿。

可以使用 kafka-replica-verification.sh 工具来验证集群分区副本的一致性。它会从指定分区的副本上获取消息，并检查所有副本是否具有相同的消息。我们必须使用正则表达式将待验证主题的名字传给它。如果不提供这个参数，它会验证所有的主题。除此之外，还需要显式地提供 broker 的地址清单。

> **副本验证对集群的影响**
>
> 副本验证工具也会对集群造成影响，因为它需要读取所有的消息。另外，它的读取过程是并行进行的，所以使用的时候要小心。

示例：对 broker 1 和 broker 2 上以 my- 开头的主题副本进行验证。

```
# kafka-replica-verification.sh --broker-list
kafka1.example.com:9092,kafka2.example.com:9092 --topic-white-list 'my-.*'
2016-11-23 18:42:08,838: verification process is started.
2016-11-23 18:42:38,789: max lag is 0 for partition [my-topic,7]
at offset 53827844 among 10 partitions
2016-11-23 18:43:08,790: max lag is 0 for partition [my-topic,7]
at offset 53827878 among 10 partitions
```

9.5 消费和生产

在使用 Kafka 时，有时候为了验证应用程序，需要手动读取消息或手动生成消息。这个时候可以借助 kafka-console-consumer.sh 和 kafka-console-producer.sh 这两个工具，它们包装了 Java 客户端，让用户不需要编写整个应用程序就可以与 Kafka 主题发生交互。

> **将结果输出到其他应用程序**
>
> 有时候，我们可能需要编写应用程序将控制台消费者和控制台生产者包装起来，用它读取消息，并把消息传给另一个应用程序去处理。这种应用程序太过脆弱，应该尽量避免编写这类应用程序。我们无法保证控制台消费者不丢失数据，也无法使用控制台生产者的所有特性，而且它发送数据的方式也很奇怪。最好的方式是直接使用 Java 客户端，或者使用其他基于 Kafka 协议实现的第三方客户端（可能是使用其他语

言开发的）。

9.5.1 控制台消费者

kafka-console-consumer.sh 工具提供了一种从一个或多个主题上读取消息的方式。消息被打印在标准输出上，消息之间以空行分隔。默认情况下，它会打印没有经过格式化的原始消息字节（使用 `DefaultFormatter`）。它有很多可选参数，其中有一些基本的参数是必选的。

> 检查工具版本
>
> 使用与 Kafka broker 相同版本的消费者客户端，这一点是非常重要的。旧版本的控制台消费者与 Zookeeper 之间不恰当的交互行为可能会影响到集群。

第一步要指定是否使用新版本的消费者，并指定 Kafka 集群的地址。如果使用的是旧版本的消费者，只需要提供 `--zookeeper` 参数，后面跟上 Kafka 集群的连接字符串。对于上面的例子来说，参数可能是 `--zookeeper zoo1.example.com:2181/kafka-cluster`。如果使用了新版本的消费者，必须使用 `--new-consumer` 和 `--broker-list`，`--broker-list` 后面需要跟上以逗号相隔的 broker 地址列表，比如 `--broker-list kafka1.example.com:9092,kafka2.example.com:9092`。

下一步要指定待读取的主题。这里有 3 个可用参数，分别是 `--topic`、`--whitelist` 和`--blacklist`。此处允许只指定一个参数。`--topic` 用于指定单个待读取的主题，`--whitelist` 和 `--blacklist` 后面跟着一个正则表达式（在命令行里可能需要转义）。与白名单正则表达式匹配的主题将会被读取，与黑名单正则表达式匹配的主题不会被读取。

示例：使用旧版消费者读取单个主题 my-topic。

```
# kafka-console-consumer.sh --zookeeper
zoo1.example.com:2181/kafka-cluster -- topic my-topic
sample message 1
sample message 2
^CProcessed a total of 2 messages
#
```

除了基本的命令行参数外，也可以把消费者的其他配置参数传给控制台消费者。可以通过两种方式来达到这个目的，这取决于需要传递的参数个数以及个人喜好。第一种方式是将配置参数写在一个文件里，然后通过 `--consumer.config CONFIGFILE` 指定配置文件，其中 CONFIGFILE 就是配置文件的全路径。另一种方式是直接在命令行以 `--consumer-property KEY=VALUE` 的格式传递一个或多个参数，其中 KEY 指参数的名字，VALUE 指参数的值。这种方式在设置消费者属性时会很有用，比如设置群组的 ID。

> 容易混淆的命令行参数
>
> 控制台消费者和控制台生产者有一个共同的参数 `--property`，千万不要将这个参数与 `--consumer-property` 和 `--producer-property` 混淆。`--property` 参数用于向消息格式化器传递配置信息，而不是给客户端本身传递配置信息。

控制台消费者的其他常用配置如下。

`--formatter CLASSNAME`

　　指定消息格式化器的类名，用于解码消息，它的默认值是 kafka.tools.DefaultFormatter。

`--from-beginning`

　　指定从最旧的偏移量开始读取数据，否则就从最新的偏移量开始读取。

`--max-messages NUM`

　　指定在退出之前最多读取 NUM 个消息。

`--partition NUM`

　　指定只读取 ID 为 NUM 的分区（需要新版本的消费者）。

01. 消息格式化器的选项

　　除了默认的消息格式化器之外，还有其他 3 种可用的格式化器。

`kafka.tools.LoggingMessageFormatter`

将消息输出到日志，而不是输出到标准的输出设备。日志级别为 INFO，并且包含了时间戳、键和值。

`kafka.tools.ChecksumMessageFormatter`

只打印消息的校验和。

`kafka.tools.NoOpMessageFormatter`

读取消息但不打印消息。

`kafka.tools.DefaultMessageFormatter`

有一些非常有用的配置选项，这些选项可以通过 `--property` 命令行参数传给它。

`print.timestamp`

如果被设为 `true`，就会打印每个消息的时间戳。

`print.key`

如果被设为 `true`，除了打印消息的值之外，还会打印消息的键。

`key.separator`

指定打印消息的键和消息的值所使用的分隔符。

`line.separator`

指定消息之间的分隔符。

`key.deserializer`

指定打印消息的键所使用的反序列化器类名。

`value.deserializer`

指定打印消息的值所使用的反序列化器类名。

反序列化类必须实现 org.apache.kafka.common.serialization.Deserializer 接口，控制台消费者会调用它们的 toString() 方法获取输出结果。一般来说，在使用 kafka_console_consumer.sh 工具之前，需要通过环境变量 CLASSPATH 将这些实现类添加到类路径里。

02. 读取偏移量主题

有时候，我们需要知道提交的消费者群组偏移量是多少，比如某个特定的群组是否在提交偏移量，或者偏移量提交的频度。这个可以通过让控制台消费者读取一个特殊的内部主题 __consumer_offsets 来实现。所有消费者的偏移量都以消息的形式写到这个主题上。为了解码这个主题的消息，需要使用 kafka.coordinator.GroupMetadataManager $OffsetsMessage Formatter 这个格式化器。

示例：从偏移量主题读取一个消息。

```
# kafka-console-consumer.sh --zookeeper
zoo1.example.com:2181/kafka-cluster -- topic __consumer_offsets
--formatter 'kafka.coordinator.GroupMetadataManager$OffsetsMessage
Formatter' --max-messages 1
[my-group-name,my-topic,0]::[OffsetMetadata[481690879,NO_METADATA]
,CommitTime 1479708539051,ExpirationTime 1480313339051]
Processed a total of 1 messages
#
```

9.5.2 控制台生产者

与控制台消费者类似，kafka-console-producer.sh 工具可以用于向 Kafka 主题写入消息。默认情况下，该工具将命令行输入的每一行视为一个消息，消息的键和值以 Tab 字符分隔（如果没有出现 Tab 字符，那么键就是 null）。

改变命令行的读取行为

如果有必要，可以使用自定义类来读取命令行输入。自定义类必须继承 kafka.common.MessageReader 类，并负责创建 ProducerRecord 对象。然后在命令行的 --

line-reader 参数后面指定这个类，并确保包含这个类的 JAR 包已经加入到类路径里。

控制台生产者有两个参数是必须指定的：--broker-list 参数指定了一个或多个 broker，它们以逗号分隔，格式为 hostname:port；另一个参数 --topic 指定了生成消息的目标主题。在生成完消息之后，需要发送一个 EOF 字符来关闭客户端。

示例：向主题 my-topic 生成两个消息。

```
# kafka-console-producer.sh --broker-list
kafka1.example.com:9092,kafka2.example.com:9092 --topic my-topic
sample message 1
sample message 2
^D
#
```

与控制台消费者一样，控制台生产者可以接受普通生产者的配置参数。这也可以通过两种方式来实现，具体用哪一种取决于你想要传递的参数个数和个人喜好。第一种方式是通过 --producer.config CONFIGFILE 指定消费者配置文件，其中 CONFIGFILE 是配置文件的全路径。另一种方式是直接在命令行以 --producer-property KEY=VALUE 的格式传递一个或多个参数，其中 KEY 指参数的名字，VALUE 指参数的值。这种方式在设置生产者属性时会很有用，比如消息批次的相关配置（如 linger.ms 或 batch.size）。

控制台生产者有很多命令行参数可用于调整它的行为。

--key-serializer CLASSNAME

　　指定消息键的编码器类名，默认是 kafka.serializer.DefaultEncoder。

--value-serializer CLASSNAME

　　指定消息值的编码器类名，默认是 kafka.serializer.DefaultEncoder。

--compression-codec STRING

　　指定生成消息所使用的压缩类型，可以是 none、gzip、snappy 或 lz4，默认值是 gzip。

`--sync`

指定以同步的方式生成消息，也就是说，在发送下一个消息之前会等待当前消息得到确认。

> 创建自定义序列化器
>
> 自定义序列化器必须继承 `kafka.serializer.Encode` 类，可以用于做一些转换操作，比如将 JSON 格式的字符串转成其他格式，如 Avro，让这些消息可以被保存到主题上。

文本行读取器的配置参数

`kafka.tools.LineMessageReader` 类负责读取标准输入，并创建消息记录。它也有一些非常有用的配置参数，可以通过 `--property` 命令行参数把这些配置参数传给控制台生产者。

`ignore.error`

如果被设为 `false`，那么在 `parse.key` 被设为 `true` 或者标准输入里没有包含键的分隔符时就会抛出异常，默认为 `true`。

`parse.key`

如果被设为 `false`，那么生成消息的键总是 `null`，默认为 `true`。

`key.separator`

指定消息键和消息值之间的分隔符，默认是 Tab 字符。

在生成消息时，LineMessageReader 使用第一个出现的 `key.separator` 作为分隔符来拆分输入。如果在分隔符之后没有其他字符，那么消息的值为空。如果输入里没有包含分隔符，或者 `parse.key` 被设为 `false`，那么消息的键就是 `null`。

9.6 客户端ACL

命令行工具 kafka-acls.sh 可以用于处理与客户端访问控制相关的问题，它的文档可以在 Apache Kafka 官方网站上找到。

9.7　不安全的操作

有一些管理操作虽然在技术上是可行的，但如果不是非常有必要，就不应该尝试那么做。比如，你正在诊断一个问题，但已经没有其他可行的办法，或者发现了一个 bug，需要一个临时解决方案。这些操作一般在文档里不会有相关的说明，而且未经证实，有可能会给应用程序带来风险。

这里会列举一些常见的操作，在紧急情况下可以使用它们。不过，一般情况下不建议执行这些操作，而且在执行之前要慎重考虑。

> 此处有危险
>
> 本节介绍的操作将涉及保存在 Zookeeper 上的元数据。除了这里提到的内容以外，不要直接修改 Zookeeper 的其他任何信息，一定要小心谨慎，因为这些操作都是很危险的。

9.7.1　移动集群控制器

每个 Kafka 集群都有一个控制器，它是运行在集群某个 broker 上的一个线程。控制器负责看管集群的操作，有时候需要将控制器从一个 broker 迁移到另一个 broker 上。例如，因为出现了某些异常，控制器虽然还在运行，但已无法提供正常的功能。这时候可以迁移控制器，但毕竟这也不是一般性的操作，所以不应该经常迁移控制器。

当前控制器将自己注册到 Zookeeper 的一个节点上，这个节点处于集群路径的最顶层，名字叫作 `/controller`。手动删除这个节点会释放当前控制器，集群将会进行新的控制器选举。

9.7.2　取消分区重分配

分区重分配的一般流程如下。

(1) 发起重分配请求（创建 Zookeeper 节点）。

(2) 集群控制器将分区添加到 broker 上。

(3) 新的 broker 开始复制分区，直到副本达到同步状态。

(4) 集群控制器从分区副本清单里移除旧的 broker。

因为分区重分配是并行进行的，所以一般情况下没有理由取消一个正在进行中的重分配任务。不过有一个例外的情况，比如在重分配进行到一半时，broker 发生了故障并且无法立即重启，这会导致重分配过程无法结束，进而妨碍其他重分配任务的进行（比如将故障 broker 的分区分配给其他 broker）。如果发生了这种情况，可以让集群忽略这个重分配任务。

移除一个进行中的分区重分配任务的步骤如下。

(1) 从 Zookeeper 上删除 `/admin/reassign_partitions` 节点。

(2) 重新选举控制器（参见 9.7.1 节）。

> **检查复制系数**
>
> 在取消进行中的分区重分配任务时，对于任何一个未完成重分配的分区来说，旧的 broker 都不会从副本清单里移除。也就是说，有些分区的复制系数会比正常的大。如果主题的分区包含不一致的复制系数，那么 broker 是不允许对其进行操作的（比如增加分区）。所以建议检查分区是否仍然可用，并确保分区的复制系数是正确的。

9.7.3　移除待删除的主题

在使用命令行工具删除主题时，命令行工具会在 Zookeeper 上创建一个节点作为删除主题的请求。在正常情况下，集群会立即执行这个请求。不过，命令行工具并不知道集群是否启用了主题删除功能。因此，如果集群没有启用主题删除功能，那么命令行工具发起的请求会一直被挂起。不过这种挂起请求是可以被移除的。

主题的删除是通过在 `/admin/delete_topic` 节点下创建一个以待删除主题名字命名的子节点来实现的。所以，删除了这些节点（不过不要删除 `/admin/delete_topic` 这个父节点），也就移除了被挂起的请求。

9.7.4　手动删除主题

如果集群禁用了主题删除功能，或者需要通过非正式的途径删除某些主题，那么可以进行手动删除。这要求在线下关闭集群里所有的 broker。

先关闭 broker

在集群还在运行的时候修改 Zookeeper 里的元数据是很危险的，这会造成集群不稳定。所以，不要在集群还在运行的时候删除或修改 Zookeeper 里的主题元数据。

从集群里手动删除主题的过程如下。

(1) 关闭集群里所有的 broker。

(2) 删除 Zookeeper 路径 /brokers/topics/TOPICNAME，注意要先删除节点下的子节点。

(3) 删除每个 broker 的分区目录，这些目录的名字可能是 `TOPICNAME-NUM`，其中 `NUM`是指分区的 ID。

(4) 重启所有的 broker。

9.8　总结

运行一个 Kafka 集群需要付出很大的努力，为了让 Kafka 保持巅峰状态，需要做大量的配置和维护。我们在这一章里介绍了 Kafka 的很多日常操作，比如经常会用到的主题管理和客户端配置；也介绍了一些用于诊断问题的复杂操作，比如检查日志片段；最后还介绍了一些不安全的操作，这些操作在特殊的情况下可以帮你解决问题。通过执行这些操作，你就可以更好地管理 Kafka 集群。

当然，如果没有进行适当的监控，管理集群就是一个不可能完成的任务。第 10 章将会讨论如何对 broker 和集群的健康状况以及操作进行监控，这样就可以知道 Kafka 的运行状态了。我们也会提供一些有关客户端（包括生产者和消费者）监控的最佳实践。

第 10 章　监控 Kafka

Kafka 应用程序包含了大量的度量指标，以至于很多人搞不清楚哪些是重要的，哪些可以置之不理。它们所涉及的范围，从简单的流量速

率度量指标到各种请求类型的时间度量指标，既有主题级别的，也有分区级别的。这些度量指标为 broker 的每一种行为提供了详细的信息，不过它们也成为了 Kafka 系统监控者的“噩梦”。

这一章将详细介绍一些常用的关键性度量指标，以及如何根据这些指标采取相应的行动，也会介绍一些用于调试问题的度量指标。不过，这不是一个完整的度量指标清单。度量指标清单会经常发生变化，而且很多度量指标只对有经验的 Kafka 开发人员有参考价值。

10.1　度量指标基础

在介绍 Kafka broker 和客户端的度量指标之前，先来讨论有关监控 Java 应用程序的基础知识，以及有关监控和告警的最佳实践。学完本章知识，读者将会对应用程序的监控有一个基本的了解，同时能明白度量指标的重要性。

10.1.1　度量指标在哪里

Kafka 提供的所有度量指标都可以通过 Java Management Extensions（JMX）接口来访问。要在外部监控系统里使用这些度量指标，最简单的方式是将负责收集度量指标的代理（agent）连接到 Kafka 上。代理作为一个单独的进程运行在监控系统里，并连接到 Kafka 的 JMX 接口上，例如使用 Nagios XI check_jmx 插件或 jmxtrans 来连接 JMX 接口；也可以直接在 Kafka 进程里运行一个 JMX 代理，然后通过 HTTP 连接访问度量指标，比如 Jolokia 或 MX4J。

本章将不会深入讨论如何设置监控代理，每一种代理都有多种使用方式。如果所在的组织没有监控 Java 应用程序的经验，那么可以考虑使用第三方的监控服务。如果采用了这种方式，那就需要从服务供应商那里购买监控服务，由他们提供监控代理、度量指标收集点、存储、图形化和告警。服务供应商可以搭建监控代理，并提供后续的服务支持。

> 找到 **JMX** 端口
>
> broker 将 JMX 端口作为整个 broker 配置信息的一部分保存在 Zookeeper 上。所以，如果要通过编程的方式访问 Kafka 的 JMX，比如管理工具需要在没有端口配置的情况下连接到 JMX，那么可以从

Zookeeper 上获取端口信息。`/brokers/ids/<ID>` 节点包含了 JSON 格式的 broker 信息，里面有 JMX 对应的主机名（`hostname`）和端口（`jmx_port`）。

10.1.2　内部或外部度量

JMX 接口提供的是内部度量指标，它们由被监控的应用程序生成。对于很多内部度量来说（比如各个请求阶段的时间），使用内部度量指标是最好的选择。没有什么能比应用程序更加了解自己了。还有一些度量指标，比如请求的整体时间、某种请求类型的可用性，可以在应用程序外部进行度量。也就是说，这些度量指标是由客户端或其他第三方应用（对于我们来说就是指 Kafka broker）提供的。另外也有一些度量指标，比如可用性（broker 是否可达）或延迟（请求需要的时间）。这些度量指标为被监控的应用程序提供了很有用的外部视图。

网站健康监控就是我们所熟知的一种外部度量。一个正常运行的 Web 服务器能够正常处理请求，它向监控系统发送度量指标，一切看起来都很好。不过，Web 服务器本身的防火墙或服务器所处网络的防火墙可能导致客户端无法连接到服务器上。负责检查网站可访问性的外部监控系统将会检测到这个问题，并发送告警。

10.1.3　应用程序健康检测

不管通过哪一种方式从 Kafka 收集监控信息，都要确保能够通过简单的健康检测来监控应用程序本身的健康状况，这可以通过两种方式来实现。

- 使用外部进程来报告 broker 的运行状态（健康检测）。
- 在 broker 停止发送度量指标时发出告警（也叫作 **stale** 度量指标）。

虽然第二种方式也是可行的，但有时候很难区分是 broker 出现了问题还是监控系统本身出现了问题。

如果监控的是 broker，可以直接连接到它的外部端口（就是客户端连接到 broker 所使用的端口），看看是否可以得到响应。如果监控的是 Kafka 客户端，就会比较复杂，需要检查进程是否处于运行状态，或者通过内部提供的方法来确定应用程序的健康状况。

10.1.4　度量指标的覆盖面

Kafka 提供了很多度量指标，如何选择合适的度量指标非常关键，特别是在基于这些度量指标定义告警的时候。太多难以确定严重程度的告警很容易让人们陷入“告警疲劳”，我们难以为每一个度量指标定义恰当的阈值并保持更新，告警的可信度也会因此而下降。

一些大覆盖面的告警用处更大。也就是说，这类告警会告诉我们某处出现了问题，然后我们去收集更多的信息，以便确定问题的细节。想象一下汽车的“检查引擎”告警灯，如果仪表盘上有 100 个不同的指示器，比如空气过滤器、油箱、排气管等，那么就会让人感到很困惑。相反，如果用一个指示器就能告诉我们汽车出现了问题，然后我们再去找出问题的其他细节，事情就会变得简单很多。这一章将介绍具有大覆盖面的度量指标，它们可以让告警变得更简单。

10.2　broker的度量指标

broker 提供了很多度量指标，它们大部分都是底层的度量，由 Kafka 开发者出于诊断或调试的目的而增加，也有为预测将来需要的信息而添加，另外还有由 Kafka 用户请求添加，以供日常操作所需。这些度量指标提供的信息几乎涵盖了 broker 的每一项功能。它们很容易造成信息过载，不过也有一些度量指标为 Kafka 的日常运行提供了必要的信息。

> 示例：谁来看着 **watcher**
>
> 很多组织使用 Kafka 收集应用程序和系统的度量指标与日志，然后供中心监控系统使用，这样可以很好地解耦应用程序和监控系统。不过，对于 Kafka 本身来说却存在一个问题，如果使用这个监控系统来监控 Kafka，那么当 Kafka 崩溃时，我们很可能无法感知到，因为监控系统的数据流也随着消失了。
>
> 有一些办法可以解决这个问题。一种方法是使用一个单独的监控系统来监控 Kafka，这个系统不依赖 Kafka 提供的数据。另一种方法是，如果有多个数据中心，可以将数据中心 A 的 Kafka 集群度量指标生成到数据中心 B 的 Kafka 集群上，反之亦然。不管怎样，要确保 Kafka 的监控和告警不依赖 Kafka 本身。

本节将从讨论非同步分区度量指标开始，介绍如何根据这些度量指标

采取行动，然后讨论其他的度量指标，以便对 broker 的度量指标有一个全面的认识。这里不会列出 broker 的所有度量指标，但会列出那些在监控 broker 和集群时必须用到的部分。在介绍客户端度量指标之前，还会针对日志展开详细的讨论。

10.2.1　非同步分区

如果说 broker 只有一个可监控的度量指标，那么它一定是指非同步分区的数量。该度量指明了作为首领的 broker 有多少个分区处于非同步状态。这个度量可以反映 Kafka 的很多内部问题，从 broker 的崩溃到资源的过度消耗。因为这个度量指标可以说明很多问题，所以当它的值大于零时，就应该想办法采取相应的行动。本章稍后会介绍更多用于诊断这类问题的度量指标。表 10-1 列出了非同步分区度量指标的详细信息。

表**10-1**：度量指标和对应的非同步分区

度量指标名称	Under-replicated partitions
JMX MBean	kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions
值区间	非负整数

如果集群里多个 broker 的非同步分区数量一直保持不变，那说明集群中的某个 broker 已经离线了。整个集群的非同步分区数量等于离线 broker 的分区数量，而且离线 broker 不会生成任何度量指标。这个时候，需要检查这个 broker 出了什么问题，并解决问题。通常有可能是硬件问题，也有可能是操作系统问题或者 Java 问题，导致进程出现中断或挂起。

> 默认的副本选举
>
> 在诊断问题之前，尝试过运行默认的副本选举（参见第 9 章）吗？ broker 在释放首领角色（发生崩溃或被关闭）之后不会自动恢复首领角色（除非启用了首领自动再均衡，不过不建议启用这个功能）。也就是说，集群里的首领副本很容易出现不均衡。运行默认的副本选举是很容易的，也很安全，所以在出现这类问题时，建议先重新选举首领，看看能否解决问题。

如果非同步分区的数量是波动的，或者虽然数量稳定但并没有 broker 离线，说明集群出现了性能问题。这类问题繁复多样，难以诊断，不

过可以通过一些步骤来缩小问题的范围。第一步，先确认问题是与单个 broker 有关还是与整个集群有关。不过有时候这个也难有定论。如果非同步分区属于单个 broker，那么这个 broker 就是问题的根源，表象是其他 broker 无法从它那里复制消息。

如果多个 broker 都出现了非同步分区，那么有可能是集群的问题，也有可能是单个 broker 的问题。这时候有可能是因为一个 broker 无法从其他 broker 那里复制数据。为了找出这个 broker，可以列出集群的所有非同步分区，并检查它们的共性。使用 kafka-topics.sh 工具（第 9 章已详细介绍过）可以获取非同步分区清单。

示例：列出集群的非同步分区。

```
# kafka-topics.sh --zookeeper zoo1.example.com:2181/kafka-cluster --descri
--under-replicated
    Topic: topicOne   Partition: 5    Leader: 1    Replicas: 1,2 Isr: 1
    Topic: topicOne   Partition: 6    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicTwo   Partition: 3    Leader: 4    Replicas: 2,4 Isr: 4
    Topic: topicTwo   Partition: 7    Leader: 5    Replicas: 5,2 Isr: 5
    Topic: topicSix   Partition: 1    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicSix   Partition: 2    Leader: 1    Replicas: 1,2 Isr: 1
    Topic: topicSix   Partition: 5    Leader: 6    Replicas: 2,6 Isr: 6
    Topic: topicSix   Partition: 7    Leader: 7    Replicas: 7,2 Isr: 7
    Topic: topicNine  Partition: 1    Leader: 1    Replicas: 1,2 Isr: 1
    Topic: topicNine  Partition: 3    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicNine  Partition: 4    Leader: 3    Replicas: 3,2 Isr: 3
    Topic: topicNine  Partition: 7    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicNine  Partition: 0    Leader: 3    Replicas: 2,3 Isr: 3
    Topic: topicNine  Partition: 5    Leader: 6    Replicas: 6,2 Isr: 6
#
```

在这个示例中，broker 2 出现在所有的副本清单里，但没有出现在所有的同步副本（ISR）清单里，所以要将注意力放在这个 broker 上。如果没有发现这样的 broker，那么问题有可能出在整个集群上。

01. 集群级别的问题

    集群问题一般分为以下两类。

    - 不均衡的负载。
    - 资源过度消耗。

    分区或首领的不均衡问题虽然解决起来有点复杂，但问题的定位是很容易的。为了诊断这个问题，需要用到 broker 的以下度

量指标。

- 分区的数量。
- 首领分区的数量。
- 主题流入字节速率。
- 主题流入消息速率。检查指标。在一个均衡的集群里，这些度量指标的数值在整个集群范围内是均等的，如表 10-2 所示。

表**10-2**：资源使用情况度量指标

Broker	分区	首领	流入字节	流出字节
1	100	50	3.56 MB/s	9.45 MB/s
2	101	49	3.66 MB/s	9.25 MB/s
3	100	50	3.23 MB/s	9.82 MB/s

也就是说，所有的 broker 几乎处理相同的流量。假设在运行了默认的副本选举之后，这些度量指标出现了很大的偏差，那说明集群的流量出现了不均衡。要解决这个问题，需要将负载较重的 broker 分区移动到负载较轻的 broker 上。这可以使用第 9 章介绍的 kafka-reassign-partitions.sh 工具来实现。

实现集群负载均衡的辅助工具

broker 本身无法在整个集群里实现自动的分区重分配。也就是说，Kafka 集群的流量均衡是一个十分费劲的过程，需要手动检查一大串度量指标，然后进行均衡的副本重分配。为了解决这个问题，有一些组织开发了自动化工具来帮助完成这个任务。例如，LinkedIn 发布了一个叫作 kafka-assigner 的工具，可以在 GitHub 的开源代码仓库 kafka-tools 里找到。Kafka 提供的企业支持服务也包含了这一功能。

Kafka 集群的另一个性能问题是容量瓶颈。有很多潜在的瓶颈会拖慢整个系统：CPU、磁盘 IO 和网络吞吐量是其中最为常见的。磁盘的使用并不在其列，因为当磁盘被填满时， broker 会在进行适当的操作之后直接崩溃。为了诊断容量问题，可以对如下一些操作系统级别的度量指标进行监控。

- CPU 使用。
- 网络输入吞吐量。
- 网络输出吞吐量。
- 磁盘平均等待时间。
- 磁盘使用百分比。

上述任何一种资源出现过度消耗，都会表现为分区的不同步。要记住，broker 的复制过程使用的也是 Kafka 客户端。如果集群的数据复制出现了问题，那么集群的用户在生产消息或读取消息时也会出现问题。所以，有必要为这些度量指标定义一个基线，并设定相应的阈值，以便在容量告急之前定位问题。随着集群流量的增长，对这些度量指标的趋势走向进行检查也是很有必要的。其中，`All Topics Bytes In Rate` 最适合用于显示集群的使用情况。

13. 主机级别的问题

如果性能问题不是出现在整个集群上，而是出现在一两个 broker 里，那么就要检查 broker 所在的主机，看看是什么导致它与集群里的其他 broker 不一样。主机级别的问题可以分为以下几类。

- 硬件问题。
- 进程冲突。
- 本地配置的不一致。

> 典型的服务器和典型的问题
>
> 当一个服务器及其操作系统承载了数千个组件时，会变得很复杂，任何一个组件都可能出现问题，导致整体的崩溃或者部分的性能衰退。本书不可能覆盖到所有的内容，关于这个话题的书已经有很多了，而且这种局面还会一直持续下去。不过，本书可以讨论一些最为常见的问题，这一小节将着重讨论运行 Linux 操作系统的服务器。

硬件问题很容易被发现，因为服务器会直接停止工作。不过，引起性能衰退的硬件问题却不那么明显。当出现这类问题时，系统仍旧保持运行，但会降低行为能力。比如内存出现了坏点，系统检测到坏点，直接跳过这个片段（可用的内存因此减

少了）。类似的问题也会发生在 CPU 上。对于这类问题，可以使用硬件提供的工具来监控硬件的健康状况，比如智能平台管理接口（IPMI）。出现问题时，可以通过 `dmesg` 查看输出到系统控制台的内核缓冲区日志。

能够导致 Kafka 性能衰退的一个比较常见的硬件问题是磁盘故障。Kafka 使用磁盘来存储消息，生产者的性能与磁盘的写入速度有直接关系。这里出现的任何偏差都会表现为生产者和复制消息者的性能问题，而后者会导致分区的不同步。因此，应该持续地监控磁盘，并在出现问题时马上进行修复。

> 一粒老鼠屎
>
> 一个 broker 的磁盘问题可能会影响到整个集群的性能。因为生产者客户端会连接到所有的 broker 上，如果操作得当，这些 broker 的分区几乎能够均等地分布在整个集群里。如果一个 broker 性能出现衰退并拖慢了处理请求的速度，就会导致生产者的回压，从而拖慢发给所有 broker 的请求。

假设你正在通过 IPMI 或其他硬件管理接口来监控磁盘的状态，与此同时，在操作系统上运行了 SMART（Self-Monitoring, Analysis and Reporting Technology，自行监控、分析和报告技术）工具来监控和测试磁盘。在故障即将发生时，它会发出告警。除此之外，还要注意查看磁盘控制器，不管是否使用了 RAID 硬件都要注意查看。很多磁盘控制器都有板载的缓存，这个缓存只在控制器和电池备份单元（BBU）正常工作时才会被使用。如果 BBU 发生故障，缓存就会被禁用，磁盘的性能就会衰退。

在网络方面，局部的故障也会带来很大的问题。有些问题是硬件引起的，比如糟糕的光缆或连接器。有些问题是配置不当造成的，比如更改了服务器或上游网络硬件的网络连接速度和双工设置。网络配置问题还有可能出现在操作系统上，比如网络缓冲区太小，或者太多的网络连接占用了大量的系统内存。在这方面，网络接口的错误数量是一个最为关键的指标。如果这个数字一直在增长，说明网络连接出现了问题。

如果硬件没有问题，那么需要注意系统里的其他应用程序，它们也会消耗系统的资源，而且有可能会给 Kafka 带来压力。它

们有可能是没有被正常安装的软件，或者一个非正常运行的进程，比如监控代理进程。对于这种情况，可以使用 `top` 工具来识别那些大量消耗 CPU 或内存的进程。

如果经过上述的检查还是找不出主机的问题根源，那么有可能是 broker 或者系统配置不一致造成的。一个服务器上运行着多个应用程序，每个应用程序有多个配置选项，要找出它们的差别真是一项艰巨的任务。这就是为什么要使用配置管理工具（如 Chef 或 Puppet）来维护操作系统和应用程序（包括 Kafka）的配置一致性。

10.2.2　broker度量指标

除了非同步分区数量外，还有其他很多 broker 级别的度量指标需要监控。虽然不一定会为所有的度量指标设定告警阈值，但它们的确提供了关于 broker 和集群的有价值的信息。它们都应该出现在监控仪表盘上。

01. 活跃控制器数量

该指标表示 broker 是否就是当前的集群控制器，其值可以是 0 或 1。如果是 1，表示 broker 就是当前的控制器。任何时候，都应该只有一个 broker 是控制器，而且这个 broker 必须一直是集群控制器。如果出现了两个控制器，说明有一个本该退出的控制器线程被阻塞了，这会导致管理任务无法正常执行，比如移动分区。为了解决这个问题，需要将这两个 broker 重启，而且不能通过正常的方式重启，因为此时它们无法被正常关闭。表 10-3 给出了活跃控制器数量度量指标的详细信息。

表**10-3**：活跃控制器数量度量指标

度量指标名称	**Active controller count**
JMX MBean	`kafka.controller:type=KafkaController,name=Ac`
值区间	0 或 1

如果集群里没有控制器，集群就无法对状态的变更作出恰当的响应，状态的变更包括主题或分区的创建和 broker 故障。这时候要注意检查为什么控制器线程没有正常运行，比如，Zookeeper 集群的网络分区就会造成这样的问题。解决底层的问题之后，重启集群里的所有 broker，重置控制器线程的状

态。

02. 请求处理器空闲率

Kafka 使用了两个线程池来处理客户端的请求：网络处理器线程池和请求处理器线程池。网络处理器线程池负责通过网络读入和写出数据。这里没有太多的工作要做，也就是说，不用太过担心这些线程会出现问题。请求处理器线程池负责处理来自客户端的请求，包括从磁盘读取消息和往磁盘写入消息。因此，broker 负载的增长对这个线程池有很大的影响。表 10-4 给出了请求处理器空闲率度量指标的详细信息。

表**10-4**：请求处理器空闲率

度量指标名称	**Request handler average idle percentage**
JMX MBean	`kafka.server:type=KafkaRequestHandlerPool,nam`
值区间	从 0 到 1 的浮点数（包括 1 在内）

智能地使用线程

这样看来，好像需要数百个请求处理器线程，但实际上，请求处理器线程数没必要超过 CPU 的核数。Kafka 在使用请求处理器时是非常智能的，它会分流需要很长时间来处理的请求。例如，当请求的配额被限定或每个生产请求需要多个确认时，Kafka 就会使用这个功能。

请求处理器平均空闲百分比这个度量指标表示请求处理器空闲时间的百分比。数值越低，说明 broker 的负载越高。经验表明，如果空闲百分比低于 20%，说明存在潜在的问题，如果低于 10%，说明出现了性能问题。除了集群的规模太小之外，还有其他两个原因会增大这个线程池的使用量。首先，线程池里没有足够的线程。一般来说，请求处理器线程的数量应该与系统的处理器核数一样（包括多线程处理器）。

另一个常见的原因是线程做了不该做的事。在 Kafka 0.10 之前，请求处理器线程负责解压传入的消息批次、验证消息、分配偏移量，并在写入磁盘之前重新压缩消息。糟糕的是，压缩方法使用了同步锁。Kafka 0.10 版本引入了一种新的格式，偏移量可以直接附加在消息批次里。也就是说，生产者在发送消

息批次之前可以设置相对的偏移量，这样 broker 就可以避免解压缩和重新压缩。如果使用了支持 0.10 版本消息格式的生产者和消费者客户端，并且把 broker 的消息格式也升级到了 0.10 版本，可以发现性能有了显著的改进。这样可以降低对请求处理器线程的消耗。

03. 主题流入字节

主题流入字节速率使用 b/s 来表示，在对 broker 接收的生产者客户端消息流量进行度量时，这个度量指标很有用。该指标可以用于确定何时该对集群进行扩展或开展其他与规模增长相关的工作。它也可以用于评估一个 broker 是否比集群里的其他 broker 接收了更多的流量，如果出现了这种情况，就需要对分区进行再均衡。表 10-5 给出了详细信息。

表**10-5**：主题流入字节度量指标

度量指标名称	Bytes in per second
JMX MBean	`kafka.server:type=BrokerTopicMetrics,name=Byt`
值区间	速率为双精度浮点数，计数为整数

因为这是第一个速率度量指标，所以有必要对它的属性进行简短的描述。所有的速率指标都提供了 7 个属性，使用哪些属性完全取决于实际的需求。它们可能是具体的数字，也有可能是基于某些时间段的平均值。如果没能恰当地使用这些指标，就无法看到 broker 真实的状况。

前两个属性与度量无关，但有助于更好地理解相应的度量指标。

`EventType`

　　这是度量的单位，在这里是“字节”。

`RateUnit`

　　这是速率的时间段，在这里是“秒”。

这两个属性表明，速率是通过 b/s 来表示的，不管它的值是基于多长的时间段算出的平均值。速率还有其他 4 个不同粒度的属性。

OneMinuteRate

前 1 分钟的平均值。

FiveMinuteRate

前 5 分钟的平均值。

FifteenMinuteRate

前 15 分钟的平均值。

MeanRate

从 broker 启动到目前为止的平均值。

OneMinuteRate 波动很快，它提供了“时间点”粒度的度量，适用于查看短期的流量走势。 MeanRate 一般不会有太大变化，它提供的是整体的流量走势。虽然有一定的用途，但一般不需要对它设置告警。FiveMinuteRate 和 FifteenMinuteRate 提供了中间粒度的度量。

除了速率属性外，速率指标还有一个 Count 属性，会在 broker 启动之后保持增长。对于这个指标来说，Count 代表了从 broker 启动以来接收到流量的字节总数。将该属性用在一个支持计数度量指标的监控系统里，就可以提供度量的完整视图，而不仅仅是平均速率。

04. 主题流出字节

主题流出字节速率与流入字节速率类似，是另一个与规模增长有关的度量指标。流出字节速率显示的是消费者从 broker 读取消息的速率。流出速率与流入速率的伸缩方式是不一样的，这要归功于 Kafka 对多消费者客户端的支持。很多 Kafka 的流出速率可以达到流入速率的 6 倍！所以，单独对流出速率进行观察和走势分析是非常重要的。表 10-6 给出了详细信息。

表**10-6**：主题流出字节度量指标

度量指标名称	**Bytes out per second**
JMX MBean	kafka.server:type=BrokerTopicMetrics,name=Byt
值区间	速率为双精度浮点数，计数为整数

- 把复制消费者包括在内

  流出速率也包括副本流量，也就是说，如果所有主题都设置了复制系数 2，那么在没有消费者客户端的情况下，流出速率与流入速率是一样的。如果有一个消费者客户端从集群读取所有的消息，那么流出速率会是流入速率的 2 倍。如果不知道这一点，光是看着这些指标就会感到很疑惑。

05. 主题流入的消息

之前介绍的字节速率以字节的方式来表示 broker 的流量，而消息速率则以每秒生成消息个数的方式来表示流量，而且不考虑消息的大小。这也是一个很有用的生产者流量增长规模度量指标。它也可以与字节速率一起用于计算消息的平均大小。与字节速率一样，该指标也能反映集群的不均衡情况。表 10-7 给出了详细信息。

表**10-7**：主题流入消息度量指标

度量指标名称	**Message in per second**
JMX MBean	kafka.server:type=BrokerTopicMetrics,name=Mes
值区间	速率为双精度浮点数，计数为整数

- 为什么没有消息的流出速率

  经常会有人问，为什么没有 broker 的“流出消息”度量指标？因为在消息被读取时，broker 将整个消息批次发送给消费者，并没有展开批次，也就不会去计算每个批次包含了多少个消息，所以，broker 也就不知道发送了多少个消息。broker 为此提供了一个度量指标叫作每秒获取次数，它指的是请求速率，而不是消息个数。

06. 分区数量

broker 的分区数量一般不会经常发生改变，它是指分配给 broker 的分区总数。它包括 broker 的每一个分区副本，不管

是首领还是跟随者。如果一个集群启用了自动创建主题的功能，那么监控这个度量指标会变得很有意思，因为你会发现，这样可以让主题的创建游离于控制之外。表 10-8 给出了详细信息。

表**10-8**：分区数量度量指标

度量指标名称	**Partition count**
JMX MBean	kafka.server:type=ReplicaManager,name=Partiti
值区间	非负整数

07. 首领数量

该度量指标表示 broker 拥有的首领分区数量。与 broker 的其他度量一样，该度量指标也应该在整个集群的 broker 上保持均等。我们需要对该指标进行周期性地检查，并适时地发出告警，即使在副本的数量和大小看起来都很完美的时候，它仍然能够显示出集群的不均衡问题。因为 broker 有可能出于各种原因释放掉一个分区的首领身份，比如 Zookeeper 会话过期，而在会话恢复之后，这个分区并不会自动拿回首领身份（除非启用了自动首领再均衡功能）。在这些情况下，该度量指标会显示较少的首领分区数，或者直接显示为零。这个时候需要运行一个默认的副本选举，重新均衡集群的首领。表 10-9 给出了详细信息。

表**10-9**：首领数量度量指标

度量指标名称	**Leader count**
JMX MBean	kafka.server:type=ReplicaManager,name=LeaderC
值区间	非负整数

可以使用该指标与分区数量一起计算出 broker 首领分区的百分比。一个均衡的集群，如果它的复制系数为 2，那么所有的 broker 都应该差不多是它们的 50% 分区的首领。如果复制系数是 3，这个百分比应该降到 33%。

08. 离线分区

与非同步分区数量一样，离线分区数量也是一个关键的度量指

标（表 10-10）。该度量只能由集群控制器提供（对于其他 broker 来说，该指标的值为零），它显示了集群里没有首领的分区数量。发生这种情况主要有两个原因。

- 包含分区副本的所有 broker 都关闭了。
- 由于消息数量不匹配，没有同步副本能够拿到首领身份（并且禁用了不完全首领选举）。

表10-10：离线分区数量度量指标

度量指标名称	Offline partiions count
JMX MBean	kafka.controller:type=KafkaController,name=Of
值区间	非负整数

在一个生产环境 Kafka 集群里，离线分区会影响生产者客户端，导致消息丢失，或者造成回压。这属于“站点宕机”问题，需要立即解决。

11. 请求度量指标

第 5 章描述了 Kafka 协议，它有多种不同的请求，每种请求都有相应的度量指标。以下的请求类型都有相应的度量指标。

- ApiVersions
- ControlledShutdown
- CreateTopics
- DeleteTopics
- DescribeGroups
- Fetch
- FetchConsumer
- FetchFollower
- GroupCoordinator
- Heartbeat
- JoinGroup
- LeaderAndIsr
- LeaveGroup
- ListGroups
- Metadata
- OffsetCommit
- OffsetFetch

- Offsets
- Produce
- SaslHandshake
- StopReplica
- SyncGroup
- UpdateMetadata

每一种请求类型都有 8 个度量指标，它们分别体现了不同请求处理阶段的细节。例如，Fetch 请求有如表 10-11 所示的度量指标。

表10-11：请求度量指标

名字	JMX MBean
Total Time	kafka.network:type=RequestMetrics,name=TotalT
Request Queue Time	kafka.network:type=RequestMetrics,name=Reques
Local Time	kafka.network:type=RequestMetrics,name=LocalT
Remote Time	kafka.network:type=RequestMetrics,name=Remote
Throttle Time	kafka.network:type=RequestMetrics,name=Thrott
Response Queue Time	kafka.network:type=RequestMetrics,name=Respon
Response Send Time	kafka.network:type=RequestMetrics,name=Respon
Requests Per Second	kafka.network:type=RequestMetrics,name=Reques

Request Per Second 是一个速率指标，前面已经介绍过它的属性，这些属性显示了在单位时间内收到并处理的请求个数。该度量提供了每种请求类型的频度，尽管很多请求类型都不是很频繁发生，比如 StopReplica 和 UpdateMetadata。

表 10-11 中的 7 个 time 指标分别为请求提供了一组百分比数值以及一个离散的 Count 属性，类似于速率度量指标。这些指标都是自 broker 启动以来开始计算的，所以在查看那些长时间没有变化的度量指标时，请记住：broker 代理运行的时间越长，数据就越稳定。它们所代表的请求处理的部分如下。

Total Time

表示 broker 花在处理请求上的时间，从收到请求开始计算，直到将响应返回给请求者。

Request Queue Time

表示请求停留在队列里的时间，从收到请求开始计算，直

到开始处理请求。

Local Time

表示首领分区花在处理请求上的时间，包括把消息写入磁盘（但不一定要冲刷）。

Remote Time

表示在请求处理完毕之前，用于等待跟随者的时间。

Throttle Time

表示暂时搁置响应的时间，以便拖慢请求者，把它们限定在客户端的配额范围内。

Response Queue Time

表示响应被发送给请求者之前停留在队列里的时间。

Response Send Time

表示实际用于发送响应的时间。

每个度量指标的属性如下。

百分位

`50thPercentile`、`75thPercentile`、`95thPercentile`、`98thPercentile`、`99thPercentile`、`999thPercentile`。

`Count`

从 broker 启动至今处理的请求数量。

`Min`

所有请求的最小值。

`Max`

所有请求的最大值。

`Mean`

所有请求的平均值。

`StdDev`

整体的请求时间标准偏差。

> 什么是百分位
>
> 百分位是一种常见的时间度量方式，尽管它们容易被人误解。一个 99 百分位度量表示，整组取样（这里指请求时间）里有 99% 的值小于度量指标的值，也就是说，有 1% 的值大于指标的值。一般情况下需要查看平均值为 99% 或 99.9% 的数值，这样就可以分辨出哪些是平均的请求，哪些是异样的请求。

在这些指标和属性中，哪些对于监控来说是比较重要的？我们至少要收集 Total Time 和 Requests Per Second 的平均值及较高的百分位（99% 或 99.9%），这样就可以获知发送请求的整体性能。如果有可能，尽量为每一种请求类型收集其他 6 种时间度量指标，这样就可以将性能问题细分到请求的各个阶段。

为时间度量指标设定告警阈值有一定的难度。例如，`fetch` 请求时间受各种因素的影响，包括客户端等待消息的时间、主题的繁忙程度，以及客户端和 broker 之间的网络连接速度。不过，对于 `Produce` 请求来说，可以为 Total Time 设定 99.9% 百分位度量基线值。与非同步分区类似，`Produce` 请求的 99.9% 百分位数值快速增长说明出现了大规模的性能问题。

10.2.3 主题和分区的度量指标

broker 的度量指标描述了 broker 的一般行为，除此之外，还有很多主题实例和分区实例的度量指标。在大型的集群里，这样的度量指标会有很多，一般情况下，不太可能将它们全部收集到一个度量指标系统里。不过，我们可以用它们来调试与客户端相关的问题。例如，主题的度量指标可以用于识别出造成集群流量大量增长的主题。我们需要提供这些度量指标，这样 Kafka 的生产者和消费者就可以使用它们。不管是否会收集这些度量指标，都有必要知道它们的用处。

表 10-12 中所列的度量指标将使用主题名称“TOPICNAME”，分区 ID 为 0。在实际中，要把主题名称和分区 ID 替换成自己的名称和 ID。

01. 主题实例的度量指标

主题实例的度量指标与之前描述的 broker 度量指标非常相似。事实上，它们之间唯一的区别在于这里指定了主题名称，也就是说，这些度量指标属于某个指定的主题。主题实例的度量指标数量取决于集群主题的数量，而且用户极有可能不会监控这些度量指标或设置告警。它们一般提供给客户端使用，客户端依此评估它们对 Kafka 的使用情况，并进行问题调试。

表**10-12**：主题实例的度量指标

名字	JMX MBean
Bytes in rate	kafka.server:type=BrokerTopicMetrics,name=Byt
Bytes out rate	kafka.server:type=BrokerTopicMetrics,name=Byt
Failed fetch rate	kafka.server:type=BrokerTopicMetrics,name=Fai
Failed produce rate	kafka.server:type=BrokerTopicMetrics,name=Fai
Messages in rate	kafka.server:type=BrokerTopicMetrics,name=Mes
Fetch request rate	kafka.server:type=BrokerTopicMetrics,name=Tot
Produce request rate	kafka.server:type=BrokerTopicMetrics,name=Tot

02. 分区实例的度量指标

分区实例的度量指标不如主题实例的度量指标那样有用。另外，它们的数量会更加庞大，因为几百个主题就可能包含数千个分区。不过不管怎样，在某些情况下，它们还是有一

定用处的。Partition size 度量指标表示分区当前在磁盘上保留的数据量（见表 10-13）。如果把它们组合在一起，就可以表示单个主题保留的数据量，作为客户端配额的依据。同一个主题的两个不同分区之间的数据量如果存在差异，说明消息并没有按照生产消息的键进行均匀分布。Log segment count 指标表示保存在磁盘上的日志片段的文件数量，可以与 Partition size 指标结合起来，用于跟踪资源的使用情况。

表**10-13**：分区实例的度量指标

名字	JMX MBean
Partition size	kafka.log:type=Log,name=Size,topic=TOPICNAME,
Log segment count	kafka.log:type=Log,name=NumLogSegments,topic=
	kafka.log:type=Log,name=LogEndOffset,topic=TO

Log end offset	
Log start offset	kafka.log:type=Log,name=LogStartOffset,topic=

Log end offset 和 Log start offset 这两个度量指标分别表示消息的最大偏移量和最小偏移量。不过需要注意的是，不能用这两个指标来推算分区的消息数量，因为日志压缩会导致偏移量“丢失”，比如包含相同键的新消息会覆盖掉旧消息。不过它们在某些情况下还是很有用的，比如在进行时间戳和偏移量映射时，就可以得到更精确的映射，消费者客户端因此可以很容易地回滚到与某个时间点相对应的偏移量（尽管可能没有 Kafka 0.10.1 里引入的基于时间的索引搜索那么重要）。

非同步分区度量指标

在分区实例的度量指标中，有一个指标用于表示分区是否处于非同步状态。一般情况下，该指标对于日常的运维起不到太大作用，因为这类指标太多了。可以直接监控 broker 的非同步分区数量，然后使用命令行工具（参见第 9 章）确定哪些分区处于非同步状态。

10.2.4　Java虚拟机监控

除了 broker 的度量指标外，还应该对服务器提供的一些标准度量进行监控，包括 Java 虚拟机（JVM）。如果 JVM 频繁发生垃圾回收，就会影响 broker 的性能，在这种情况下，就应该得到告警。JVM 的度量指标还能告诉我们为什么 broker 下游的度量指标会发生变化。

01. 垃圾回收

对于 JVM 来说，最需要监控的是垃圾回收（GC）的状态。需要监控哪些 MBean 取决于具体的 Java 运行时（JRE）和垃圾回收器的设置。如果 JRE 使用了 Oracle Java 1.8，并使用了 G1 垃圾回收器，那么需要监控的 MBean 如表 10-14 所示。

表10-14：G1垃圾回收器度量指标

名字	JMX MBean
Full GC cycles	java.lang:type=GarbageCollector,name=G1 Old Generation
Yong GC cycles	java.lang:type=GarbageCollector,name=G1 Young Generation

在垃圾回收语义里，Old 和 Full 的意思是一样的。我们需要监控这两个指标的 `Collection Count` 和 `CollectionTime` 属性。`CollectionCount` 表示从 JVM 启动开始算起的垃圾回收次数，`CollectionTime` 表示从 JVM 启动开始算起的垃圾回收时间，以 ms 为单位。因为这些属性是计数器，所以它们可以告诉我们发生 GC 的次数和花费在 GC 上的时间，还可以用于计算平均每次 GC 花费的时间，不过在通常的运维中，这样做并没有多大用处。

这些指标还有一个 `LastGcInfo` 属性。这是一个由 5 个字段组成的组合值，用于提供最后一次 GC 的信息。其中最重要的一个值是 `duration` 值，它以 ms 为单位，展示最后一次 GC 花费的时间。其他几个值（`GcThreadCount`、`id`、`startTime` 和 `endTime`）虽然也会提供一些信息，但用处不大。不过要注意的是，我们无法通过该属性查看到每一次 GC 的信息，因为年轻代 GC 发生得很频繁。

02. **Java** 操作系统监控

JVM 通过 `java.lang:type=OperatingSystem` 提供了一些操作系统的信息，不过这些指标的信息量很有限，并不能告知操作系统的完整状况。这些指标有两个比较有用但在操作系统里难以收集到的属性——`MaxFileDescriptorCount` 和 `OpenFileDescriptorCount`。`MaxFileDescriptorCount` 展示 JVM 能够打开的文件描述符（FD）数量的最大值，`Open FileDescriptorCount` 展示目前已经打开的文件描述符数量。每个日志片段和网络连接都会打开一个文件描述符，所以它们的数量增长得很快。如果网络连接不能被正常关闭，那么 broker 很快就会把文件描述符用完。

10.2.5　操作系统监控

JVM 并不能告诉用户所有与操作系统有关的信息，因此，用户不仅要收集 broker 的度量指标，也需要收集操作系统的度量指标。大多数监控系统都会提供代理，用于收集更多有关操作系统的信息。用户需要监控 CPU 的使用、内存的使用、磁盘的使用、磁盘 IO 和网络的使用情况。

在 CPU 方面，需要监控平均系统负载。系统负载是一个独立的数值，它展示处理器的相对使用情况。另外，根据类型来捕捉 CPU 的

使用百分比也是很有用的。根据收集方法和操作系统的不同，可以使用如下列出的 CPU 百分比数值（使用了缩写），既可以使用其中的一部分，也可以使用全部。

`us`

用户空间使用的时间。

`sy`

内核空间使用的时间。

`ni`

低优先级进程使用的时间。

`id`

空闲时间。

`wa`

磁盘等待时间。

`hi`

处理硬件中断的时间。

`si`

处理软件中断的时间。

`st`

等待管理程序的时间。

 什么是系统负载

尽管很多人都知道系统负载是对 CPU 使用情况的度量，但他们并不知道度量是如何进行的。平均负载是指等待处理器执行的线程数。在 Linux 系统里，它还包括处于不可中断睡眠状态的线程，比如磁盘等待。负载使用 3 个数值来表示，分别是前 1 分钟的平均

值，前 5 分钟的平均值和前 15 分钟的平均值。在单 CPU 系统里，数值 1 表示系统负载达到了 100%，此时总会存在一个等待执行的线程。而在多 CPU 系统里，如果负载达到 100%，那么负载的数值与 CPU 的核数相等。例如，如果系统里有 24 个处理器，那么负载 100% 表示负载数值为 24。

对于 broker 来说，跟踪 CPU 的使用情况是很有必要的，因为它们在处理请求时使用了大量的 CPU 时间。而内存使用情况的跟踪就显得没有那么重要了，因为运行 Kafka 并不需要太大的内存。它会使用堆外的一小部分内存来实现压缩功能，其余大部分内存则用于缓存。不过，我们还是要对内存使用情况进行跟踪，确保其他的应用不会影响到 broker。可以通过监控总内存空间和可用交换内存空间来确保内存交换空间不会被占用。

对于 Kafka 来说，磁盘是最重要的子系统。所有的消息都保存在磁盘上，所以 Kafka 的性能严重依赖磁盘的性能。我们需要对磁盘空间和索引节点进行监控，确保磁盘空间不会被用光。对于保存数据的分区来说就更是如此。对磁盘 IO 进行监控也是很有必要的，它们揭示了磁盘的运行效率。我们需要监控磁盘的每秒种读写速度、读写平均队列大小、平均等待时间和磁盘的使用百分比。

最后，还需要监控 broker 的网络使用情况。简单地说，就是指流入和流出的网络流量，一般使用 b/s 来表示。要记住，在没有消费者时，1 个流入比特对应 1 个或多个流出比特，这个数字与主题的复制系数相等。根据消费者的实际数量，流入流量很容易比输出流量高出一个数量级。在设置告警阈值时要切记这一点。

10.2.6　日志

在讨论监控时，如果不涉及日志，那么这个监控就是不完整的。与其他应用程序一样， Kafka 的 broker 也可以被配置成定期向磁盘写入日志。为了能够从日志里获得有用的信息，选择合适的日志和日志级别是很重要的。通过记录 INFO 级别的日志，可以捕捉到很多有关 broker 运行状态的重要信息。为了获得一系列清晰的日志文件，很有必要对日志进行分类。

可以考虑使用两种日志。第一种是 `kafka.controller`，可以将它设置为 `INFO` 级别。这个日志用于记录集群控制器的信息。在任何时候，集群里都只有一个控制器，因此只有一个 broker 会使用这个日志。日志里包含了主题的创建和修改操作、broker 状态的变更，以及

集群的活动，比如默认的副本选举和分区的移动。另一个日志是 kafka.server.ClientQuotaManager，也可以将它设置为 INFO 级别。这个日志用于记录与生产和消费配额活动相关的信息。因为这些信息很有用，所以最好不要把它们记录在 broker 的主日志文件里。

在调试问题时，还有一些日志也很有用，比如 kafka.request.logger，可以将它设置为 DEBUG 或 TRACE 级别。这个日志包含了发送给 broker 的每一个请求的详细信息。如果日志级别被设置为 DEBUG，那么它将包含连接端点、请求时间和概要信息。如果日志级别被设置为 TRACE，那么它将包含主题和分区的信息，以及除消息体之外的所有与请求相关的信息。不管被设置成什么级别，这个日志会产生大量的数据，所以如果不是出于调试的目的，不建议启用这个日志。

日志压缩线程的运行状态也是一个有用的信息。不过，这些线程并没有单独的度量指标，一个分区的压缩失败有可能造成压缩线程的整体崩溃，而且是悄然发生的。启用 kafka.log.LogCleaner、kafka.log.Cleaner 和 kafka.log.LogCleanerManager 这些日志，并把它们设置为 DEBUG 级别，就可以输出日志压缩线程的运行状态。这些日志包含了每个被压缩分区的大小和消息个数。一般情况下，这些日志的数量不会很大。也就是说，默认启用这些日志并不会带来什么麻烦。

10.3　客户端监控

所有的应用程序都需要监控，包括那些使用了 Kafka 客户端的应用程序。不管是生产者还是消费者，它们都有特定的度量指标需要监控。本节将主要介绍如何监控官方的 Java 客户端，其他第三方的客户端应该也有自己的度量指标。

10.3.1　生产者度量指标

新版本 Kafka 生产者客户端的度量指标经过调整变得更加简洁，只用了少量的 MBean。相反，之前版本的客户端（不再受支持的版本）使用了大量的 MBean，而且度量指标包含了大量的细节（提供了大量的百分位和各种移动平均数）。这些度量指标提供了很大的覆盖面，但这样会让跟踪异常情况变得更加困难。

生产者度量指标的 MBean 名字里都包含了生产者的客户端 ID。在下

面的示例里，客户端 ID 使用 `CLIENTID` 表示，broker ID 使用 `BROKERID` 表示，主题的名字使用 `TOPICNAME` 表示，如表 10-15 所示。

表10-15：Kafka生产者度量指标MBean

名称	JMX MBean
Overall producer	kafka.producer:type=producer-metrics,client-id=CLIENTID
Per-broker	kafka.producer:type=producer-node-metrics,client-id=CLIENTID,node-id=node-BROKERID
Per-topic	kafka.producer:type=producer-topic-metrics,client-id=CLIENTID,topic=TOPICNAME

上表中列出的每一个 MBean 都有多个属性用于描述生产者的状态。下面列出了用处最大的几个属性。在继续了解这些属性之前，建议先通过阅读第 3 章了解生产者的工作原理。

01. 生产者整体度量指标

    生产者整体度量指标提供的属性描述了生产者各个方面的信息，从消息批次的大小到内存缓冲区的使用情况。虽然这些度量在调试的时候很有用，不过在实际应用中只会用到少数的几个，而在这几个度量里，也只有一部分需要进行监控和告警。下面将讨论一些平均数度量指标（以 `-avg` 结尾），它们都有相应的最大值（以 `-max` 结尾），不过这些最大值的用处很有限。

    `record-error-rate` 是一个完全有必要对其设置告警的属性。这个指标的值一般情况下都是零，如果它的值比零大，说明生产者正在丢弃无法发送的消息。生产者配置了重试次数和回退策略，如果重试次数达到了上限，消息（记录）就会被丢弃。我们可以跟踪另外一个属性 `record-retry-rate`，不过该属性不如 `record-error-rate` 那么重要，因为重试是很正常的行为。

    我们可以对 `request-latency-avg` 设置告警，它表示发送一个生产者请求到 broker 所需要的平均时间。在运维过程中，应该为该指标建立一个基线，并设定告警阈值。请求延迟的增加说明生产者请求正在变慢。有可能是网络出现了问题，也有可能是 broker 出现了问题。不管是哪一种情况，都会导致生产者端发生回压，并引发应用程序的其他问题。

除了这些关键性的度量指标外，如果能够知道生产者发送的消息流量就更好了。有 3 个属性提供了 3 个不同的视图：`outgoing-byte-rate` 表示每秒钟消息的字节数，`record-send-rate` 表示每秒钟消息的数量，`request-rate` 表示每秒钟生产者发送给 broker 的请求数。单个请求可以包含一个或多个批次，单个批次可以包含一个或多个消息，每个消息由多个字节组成。把这些指标显示在仪表盘上，可以跟踪到生产者是如何使用 Kafka 的。

> 为什么不使用 **`ProducerRequestMetrics`**
>
> `ProducerRequestMetrics` MBean 提供了请求延迟的百分位和请求速率的移动平均数，但为什么不建议使用呢？问题在于，这个度量指标是为每个生产者线程提供的。如果应用程序出于性能方面的考虑，使用了多个生产者线程，那么就很难对这些指标进行聚合。所以，使用生产者整体指标所提供的属性是一种更为有效的方式。

可以借助一些度量指标来更好地理解字节、记录、请求和批次之间的关系，这些指标描述了这些实体的大小。`request-size-avg` 表示生产者发送请求的平均字节数，`batch-size-avg` 表示单个消息批次（根据定义，批次包含了属于同一个分区的多个消息）的平均字节数，`record-size-avg` 表示单个消息的平均字节数。对于单主题的生产者来说，它们提供了有关生产消息的有用信息。而对于多主题的生产者来说，比如 MirrorMaker，它们提供的信息就不是很有用。除了这 3 个指标外，还有另外一个指标 `records-per-request-avg`，它表示在生产者的单个请求里所包含的消息平均个数。

最后一个值得推荐的指标是 `record-queue-time-avg`，它表示消息在发送给 Kafka 之前在生产者客户端等待的平均毫秒数。应用程序在使用生产者客户端发送消息（调用 `send` 方法）之后，生产者会一直等待，直到以下任一情况发生。

- 生产者客户端有足够多的消息来填充批次（根据 `max.partition.bytes` 的配置）。
- 距离上一次发送批次已经有足够长的时间（根据 `linger.ms` 的配置）。

这两种情况都会促使生产者客户端关闭当前批次，然后把它发送给 broker。对于繁忙的主题来说，一般会发生第一种情况，而对于比较慢的主题来说，一般会发生第二种情况。`record-queue-time-avg` 用于度量生产消息所使用的时间，因此，可以通过调优上述两个参数来满足应用程序的延迟需求。

04. **Per-broker** 和 **Per-topic** 度量指标

除了生产者整体的度量指标外，还有一些 MBean 为每个 broker 连接和每个主题提供了有限的属性集合。在调试问题时，这些度量会很有用，但不建议对它们进行常规的监控。这些 MBean 属性的命名与整体度量指标的命名是一样的，所表示的含义也是一样的（只不过它们是作用在特定的 broker 或特定的主题上）。

在 broker 实例的度量指标里，`request-latency-avg` 是比较有用的一个，因为它一般比较稳定（在消息批次比较稳定的情况下），而且能够显示 broker 的连接问题。其他的属性，比如 `outgoing-byte-rate` 和 `request-latency-avg`，它们会因为 broker 所包含分区的不同而有所不同。也就是说，这些度量会随着时间发生变化，完全取决于集群的状态。

主题实例的度量指标比 broker 实例的度量指标要有趣得多，不过只有在使用多主题的生产者时，它们才能派上用场。当然，也只有在生产者不会消费过多主题的情况下，才有必要对这些指标进行常规的监控。例如，MirrorMaker 有可能会向成百上千的主题生成消息。我们无法逐个检查这些指标，也几乎不可能为它们设置合理的告警阈值。与 broker 实例的度量指标一样，主题实例的度量指标一般用于诊断问题。例如，可以通过 `record-send-rate` 和 `record-error-rate` 这两个属性将丢弃的消息隔离到特定的主题上。另外，`byte-rate` 表示主题整体的每秒消息字节数。

10.3.2　消费者度量指标

与新版本的生产者客户端类似，新版本的消费者客户端将大量的度量指标属性塞进了少数的几个 MBean 里，如表 10-6 所示。消费者客户端也作出了权衡，去掉了延迟百分位和速率移动平均数。因为消费者读取消息的逻辑比生产者发送消息的逻辑复杂，所以消费者的度量指标也会更多。具体请参考表 10-16。

表10-16：Kafka消费者度量指标MBean

名称	JMX MBean
Overall consumer	kafka.consumer:type=consumer-metrics,client-id=CLIENTID
Fetch manager	kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID
Per-topic	kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID,topic=TOPICNAME
Per-broker	kafka.consumer:type=consumer-node-metrics,client-id=CLIENTID,node-id=node-BROKERID
Coordinator	kafka.consumer:type=consumer-coordinator-metrics,client-id=CLIENTID

01. **Fetch Manager** 度量指标

在消费者客户端，消费者整体度量指标 MBean 的用处不是很大，因为有用的指标都聚集在 Fetch Manager MBean 里。消费者 MBean 包含了与网络底层运行情况有关的指标，而 Fetch Manager MBean 则包含了与字节、请求和消息速率有关的指标。与生产者客户端不同的是，用户可以查看消费者所提供的度量指标，但对它们设置告警是没有意义的。

对于 Fetch Manager 来说，`fetch-latency-avg` 可能是一个需要对其进行监控和设置告警的指标。与生产者的 `request-latency-avg` 类似，该指标表示从消费者向 broker 发送请求所需要的时间。请求的延迟通过消费者的 `fetch.min.bytes` 和 `fetch.max.wait.ms` 这两个参数进行控制。一个缓慢的主题会出现不稳定的延迟，有时候 broker 响应很快（有可用消息的时候），有时候甚至无法在 `fetch.max.wait.ms` 规定的时间内完成响应（没有可用消息的时候）。如果主题有相对稳定和足够的消息流量，那么查看这个指标或许会更有意义。

为什么不用延时（**Lag**）

我们建议对所有消费者的延时情况进行监控，但为什么不建议对 Fetch Manager 的 `records-lag-max` 属性进行监控呢？该属性表示落后最多的分区的延时情况（落后 broker 的消息数量）。

这里有两方面的原因：一方面是因为它只显示了单个分区的延时，另一方面是因为它依赖消费者的某些特定功能。如果没有其他的选择，

> 那么可以考虑将该属性作为度量延时的指标，并为其设置告警。不过，最佳实践应该是使用外部的延时监控，稍后会介绍更多的内容。

要想知道消费者客户端处理了多少消息流量，可以使用 `bytes-consumed-rate` 或 `records-consumed-rate`，或者同时使用它们两个。它们分别表示客户端每秒读取的消息字节数和每秒读取的消息个数。有些人为它们设置了最小值告警阈值，当消费者工作负载不足时，他们就会收到告警。不过需要注意的是，消费者和生产者之间是没有耦合的，因此不能从消费者端去推测生产者端的行为模式。

Fetch Manager 也提供了一些度量指标，有助于理解字节、消息和请求之间的关系。`fetch-rate` 表示消费者每秒发出请求的数量，`fetch-size-avg` 表示这些请求的平均字节数，`records-per-request-avg` 表示每个请求的平均消息个数。要注意的是，消费者并没有提供与生产者相对应的 `record-size-avg`，所以无法知道消息的平均大小。如果这个很重要，那么可以参考其他指标，或者在应用程序接收到消息之后计算出消息的平均大小。

02. **Per-broker 和 Per-topic 度量指标**

与生产者客户端类似，消费者客户端也为每个 broker 连接和每个主题提供了很多度量指标，它们可以用于诊断问题，但不建议对它们进行常规的监控。与 Fetch Manager 一样， broker 的 `request-latency-avg` 属性用处也很有限，它取决于主题的消息流量。`incoming-byte-rate` 和 `request-rate` 分别表示 broker 每秒读取的消息字节数和每秒请求数。它们可以用于诊断消费者与 broker 之间的连接问题。

在读取多个主题时，消费者客户端所提供的主题实例的度量指标会很有用。如果只是读取单个主题，那么这些指标就与 Fetch Manager 的指标一样，没有必要去收集它们。但如果客户端读取了大量的主题，比如 MirrorMaker，那么查看这些指标就会变得很困难。如果打算收集这些指标，可以考虑收集最重要的 3 个：`bytes-consumed-rate`、`records-consumed-rate` 和 `fetch-size-avg`。`bytes-consumed-rate` 表示从某个主题上每秒读取的消息字节数，`records-consumed-rate` 表示每秒读取的消息个数，`fetch-size-avg` 表示每个请求的消息平均字节数。

03. **Coordinator 度量指标**

正如第 4 章所述，消费者客户端以群组的方式运行。群组里会发生一些需要协调的活动，比如新成员的加入或者向 broker 发送心跳来保持群组的成员关系。消费者协调器负责处理这些协调工作，作为消费者客户端的组成部分，它也维护着自己的一组度量指标。与其他度量指标一样，Coordinator 度量指标也提供了很多数字，但是其中只有关键的一小部分需要进行常规的监控。

消费者群组在进行同步时，可能会造成消费者的停顿。当群组里的消费者在协商哪些分区应该由哪些消费者来读取时，就会发生这种情况。停顿的时间长短取决于分区的数量。协调器提供了 `sync-time-avg` 属性，用于表示同步活动所使用的平均毫秒数。`sync-rate` 属性也很有用，表示每秒钟群组发生的同步次数。对于一个稳定的消费者群组来说，这个数字在多数时候都是零。

消费者需要通过提交偏移量来作为读取进度的检查点，消费者可以基于固定时间间隔自动提交偏移量，也可以通过应用程序代码手动提交偏移量。提交偏移量也是一种生成消息的请求（不过它们有自己的请求类型），提交的偏移量就是消息，并被生成到一个特定的主题上。协调器提供了 `commit-latency-avg` 属性，表示提交偏移量所需要的平均时间。与生产者里的请求延时一样，我们也需要监控该指标，为它设定一个预期的基线，并设置合理的告警阈值。

Coordinator 度量指标的最后一个属性是 `assigned-partitions`，表示分配给消费者客户端（群组里的单个实例）的分区数量。该属性之所以有用，是因为可以通过在整个群组内比较各个实例的值，从而知道群组的负载是否均衡。我们可以使用该属性来识别因协调器的分区分配算法所导致的负载不均衡问题。

10.3.3　配额

Kafka 可以对客户端的请求进行限流，防止客户端拖垮整个集群。对于消费者和生产者客户端来说，这都是可配的，可以使用每秒钟允许单个客户端访问单个 broker 的流量字节数来表示。它有一个 broker 级别的默认值，客户端可以对其进行覆盖。当 broker 发现客户端的流量已经超出配额时，它就会暂缓向客户端返回响应，等待足够长的

时间，直到客户端流量降到配额以下。

broker 并不会在响应消息里提供客户端被限流的错误码，也就是说，对于应用程序来说，如果不监控这些指标，可能就不知道发生了限流。需要监控的指标如表 10-17 所示。

表**10-17**：需要监控的度量指标

客户端	MBean
消费者	kafka.consumer:type=consumer-fetch-manager-metrics,client-id=CLIENTID 的属性 fetch-throttle-time-avg
生产者	kafka.producer:type=producer-metrics,client-id=CLIENTID 的属性 produce-throttle-time-avg

默认情况下，broker 不会开启配额功能。不过不管有没有使用配额，监控这些指标总是没有问题的。况且，它们有可能在未来某个时刻被启用，而且从一开始就监控它们要比在后期添加更加容易。

10.4　延时监控

对于消费者来说，最需要被监控的指标是消费者的延时。它表示分区最后一个消息和消费者最后读取的消息之间相差的消息个数。在之前的小节里已经说过，在这里，使用外部监控要比使用客户端自己的监控好得多。虽然消费者提供了延时指标，但该指标存在一个问题，它只表示单个分区的延时，也就是具有最大延时的那个分区，所以它不能准确地表示消费者的延时。另外，它需要消费者做一些额外的操作，因为该指标是由消费者对每个发出的请求进行计算得出的。如果消费者发生崩溃或者离线，那么该指标要么不准确，要么不可用。

监控消费者延时最好的办法是使用外部进程，它能够观察 broker 的分区状态，跟踪最近消息的偏移量，也能观察消费者的状态，跟踪消费者提交的最新偏移量。这种方式提供了一种持续更新的客观视图，而且不依赖消费者的状态。我们需要在每一个分区上进行这种检查。对于大型消费者来说，比如 MirrorMaker，可能意味着要监控成千上万个分区。

第 9 章介绍过如何使用命令行工具获取消费者群组的信息，包括提交的偏移量和延时。如果直接监控由工具提供的延时信息，会存在一些问题。首先，需要为每个分区定义合理的延时。每小时接收 100 个消息的主题和每秒钟接收 10 万个消息的主题，它们的阈值是不一样

的。其次，必须能够将延时指标导入到监控系统，并为它们设置告警。如果有一个消费者群组读取了 1500 个主题，这些主题的分区数量超过了 10 万个，那将是一项令人望而生畏的任务。

可以使用 Burrow 来完成这项工作。Burrow 是一个开源的应用程序，最初由 LinkedIn 开发。它收集集群消费者群组的信息，并为每个群组计算出一个单独的状态，告诉我们群组是否运行正常、是否落后、速度是否变慢或者是否已经停止工作，以此来完成对消费者状态的监控。它不需要通过监控群组的进度来获得阈值，不过用户仍然可以从中获得消息的延时数量。LinkedIn 工程博客上记录了有关 Burrow 工作原理的讨论。Burrow 可以用于监控集群所有的消费者，也可以用于监控多个集群，而且可以很容易被集成到现有的监控和告警系统里。

如果没有其他选择，消费者客户端的 `records-lag-max` 指标提供了有关消费者状态的部分视图。不过，仍然强烈建议使用像 Burrow 这样的外部监控系统。

10.5　端到端监控

我们推荐使用的另一种外部监控系统是端到端的监控系统，它为 Kafka 集群的健康状态提供了一种客户端视图。消费者和生产者客户端有一些度量指标能够说明集群可能出现了问题，但这里有猜想的成分，因为延时的增加有可能是由客户端、网络或 Kafka 本身引起的。另外，用户原本的工作可能只是管理 Kafka 集群，但现在也需要监控客户端。现在只需要回答以下两个简单的问题。

- 可以向 Kafka 集群生成消息吗？
- 可以从 Kafka 集群读取消息吗？

理想情况下，用户可能会希望每一个主题都允许这些操作，但要为此向每一个主题注入人为的流量是不合理的。所以，可以考虑是否每个 broker 都允许这些操作，而这正是 Kafka Monitor 要做的事情。该工具由 LinkedIn 的 Kafka 团队开发并开源，它持续地向一个横跨集群所有 broker 的主题生成消息，并读取这些消息。它对每个 broker 的生产请求和读取请求的可用性进行度量，包括从生产到读取的整体延时。这种类型的监控对于验证 Kafka 集群的运行状态来说是非常有价值的，因为 Kafka broker 本身无法告知客户端是否能够正常使用集群。

10.6　总结

监控是运行 Kafka 的一个重要组成部分，这也是为什么有那么多的团队在这上面花费了那么多时间。很多组织使用 Kafka 处理千万亿字节级别的数据流。确保数据的持续性和不丢失消息是一个关键性的业务需求。作为 Kafka 集群的运维人员，我们的目标是成为最了解集群状态的人。同时，要协助用户监控他们的应用程序。

本章介绍了如何监控 Java 应用程序，特别是 Kafka 应用程序。首先介绍了 broker 的一些度量指标，接着介绍了 Java 和操作系统的监控以及日志，然后详细地介绍了 Kafka 客户端的监控，包括配额的监控，最后讨论了如何使用外部监控系统进行消费者延时监控以及如何进行端到端的集群可用性监控。本章虽然没有列出所有可用的度量指标，但已经涵盖了最为关键的部分。

第 11 章　流式处理

Kafka 一般被认为是一个强大的消息总线，可以传递事件流，但没有处理和转换事件的能力。Kafka 可靠的传递能力让它成为流式处理系统完美的数据来源。很多基于 Kafka 构建的流式处理系统都将 Kafka 作为唯一可靠的数据来源，如 Apache Storm、Apache Spark Streaming、Apache Flink、Apache Samza 等。

有时候，行业分析师声称这些流式处理系统与那些已经存在了二十多年的复杂事件处理（CEP）系统没有什么两样。但是流式处理系统却很成功，而只有寥寥可数的系统在采用 CEP，他们为此感到很惊讶。笔者认为，CEP 的主要问题在于缺少事件流的处理能力。随着 Kafka 越来越流行，最初作为简单的消息总线，后来成为一种数据集成系统。很多公司的系统里都包含了大量有价值的数据流，它们井然有序地存在了很长时间，好像在等待一个流式处理框架的出现。换句话说，在出现数据库之前，数据的处理是一项艰巨的任务。类似地，因为缺少能够提供可靠存储和集成的流式平台，流式处理的发展也受到了阻碍。

从 0.10.0 版本开始，Kafka 不仅为每一个流行的流式处理框架提供了可靠的数据来源，还提供了一个强大的流式处理类库，并将其作为客户端类库的一部分。这样，开发人员就可以在应用程序里读取、处理

和生成事件，而不需要再依赖外部的处理框架。

本章将以解释什么是流式处理作为开头（因为这个术语经常被人误解），然后讨论流式处理的一些基本概念和流式处理系统常用的设计模式，然后深入介绍 Kafka 的流式处理类库，包括它的目标和架构，接着将会给出一个示例，介绍如何使用 Kafka Streams（以下简称 Streams）来计算股价移动平均数，最后讨论流式处理的其他使用场景，并在结束部分列出为 Kafka 选择流式处理框架（如果有的话）的参考标准。本章主要是简单地介绍流式处理，并不会涵盖 Streams 的每一个特性，也不会对每一个现有的流式处理框架进行比较，因为光这些内容就可以单独写成一本甚至好几本书了。

11.1 什么是流式处理

人们对流式处理的理解非常混乱。因为有太多关于流式处理的定义，它们混淆了实现细节、性能需求、数据模型和软件工程的各个方面。笔者亲眼目睹了发生在关系型数据库上的类似窘境，关系模型的抽象定义总是夹杂了数据库引擎的实现细节和特定局限性。

流式处理领域还处在发展阶段，有一些流行的实现方案，其处理方式可能很特别，或者有特定的局限，但这并不能说明它们的实现细节就是流式处理固有的组成部分。

先来看看什么是数据流（也被称为“事件流”或“流数据”）。首先，数据流是无边界数据集的抽象表示。无边界意味着无限和持续增长。无边界数据集之所以是无限的，是因为随着时间的推移，新的记录会不断加入进来。这个定义已经被包括 Google 和 Amazon 在内的大部分公司所采纳。

这个简单的模型（事件流）可以表示很多业务活动，比如信用卡交易、股票交易、包裹递送、流经交换机的网络事件、制造商设备传感器发出的事件、发送出去的邮件、游戏里物体的移动，等等。这个清单是无穷无尽的，因为几乎每一件事情都可以被看成事件的序列。

除了没有边界外，事件流模型还有其他一些属性。

事件流是有序的

　　事件的发生总是有个先后顺序。以金融活动事件为例，先将钱存进账户后再花钱，这与先花钱再还钱的次序是完全不一样的。后者会出现透支，而前者不会。这是事件流与数据库表的不同点之一。数据

库表里的记录是无序的，而 SQL 语法中的 order by 并不是关系模型的组成部分，它是为了报表查询而添加的。

不可变的数据记录

　　事件一旦发生，就不能被改变。一个金融交易被取消，并不是说它就消失了，相反，这需要往事件流里添加一个额外的事件，表示前一个交易的取消操作。顾客的一次退货并不意味着之前的销售记录被删除，相反，退货行为被当成一个额外的事件记录下来。这是数据流与数据表之间的另一个不同点——可以删除和修改数据表里的记录，但这些操作只不过是发生在数据库里的事务，这些事务可以被看成事件流。假设你对数据库的二进制日志（bin log）、预写式日志（WAL）和重做日志（redo log）的概念都很熟悉，那么就会知道，如果往数据库表插入一条记录，然后将其删除，表里就不会再有这条记录。但重做日志里包含了两个事务：插入事务和删除事务。

事件流是可重播的

　　这是事件流非常有价值的一个属性。用户可以很容易地找出那些不可重播的流（流经套接字的 TCP 数据包就是不可重播的），但对于大多数业务来说，重播发生在几个月前（甚至几年前）的原始事件流是一个很重要的需求。可能是为了尝试使用新的分析方法纠正过去的错误，或是为了进行审计。这也就是为什么我们相信 Kafka 能够让现代业务领域的流式处理大获成功——可以借助 Kafka 来捕捉和重播事件流。如果没有这项能力，流式处理充其量只是数据科学实验室里的一个玩具而已。

如果事件流的定义里没有提到事件所包含的数据和每秒钟的事件数量，那么它就变得毫无意义。不同系统之间的数据是不一样的，事件可以很小（有时候只有几个字节），也可以很大（包含很多消息头的 XML 消息），它们可以是完全非结构化的键值对，可以是半结构化的 JSON，也可以是结构化的 Avro 或 Protobuf。虽然数据流经常被视为“大数据”，并且包含了每秒钟数百万的事件，不过这里所讨论的技术同样适用（通常是更加适用）于小一点的事件流，可能每秒钟甚至每分钟只有几个事件。

知道什么是事件流以后，是时候了解“流式处理”的真正含义了。流式处理是指实时地处理一个或多个事件流。流式处理是一种编程范式，就像请求与响应范式和批处理范式那样。下面将对这 3 种范式进行比较，以便更好地理解如何在软件架构中应用流式处理。

请求与响应

这是延迟最小的一种范式，响应时间处于亚毫秒到毫秒之间，而且响应时间一般非常稳定。这种处理模式一般是阻塞的，应用程序向处理系统发出请求，然后等待响应。在数据库领域，这种范式就是线上交易处理（OLTP）。销售点（POS）系统、信用卡处理系统和基于时间的追踪系统一般都使用这种范式。

批处理

这种范式具有高延迟和高吞吐量的特点。处理系统按照设定的时间启动处理进程，比如每天的下午两点开始启动，每小时启动一次等。它读取所有的输入数据（从上一次执行之后的所有可用数据，或者从月初开始的所有数据等），输出结果，然后等待下一次启动。处理时间从几分钟到几小时不等，并且用户从结果里读到的都是旧数据。在数据库领域，它们就是数据仓库（DWH）或商业智能（BI）系统。它们每天加载巨大批次的数据，并生成报表，用户在下一次加载数据之前看到的都是相同的报表。从规模上来说，这种范式既高效又经济。但在近几年，为了能够更及时、高效地作出决策，业务要求在更短的时间内能提供可用的数据，这就给那些为探索规模经济而开发却无法提供低延迟报表的系统带来了巨大的压力。

流式处理

这种范式介于上述两者之间。大部分的业务不要求亚毫秒级的响应，不过也接受不了要等到第二天才知道结果。大部分业务流程都是持续进行的，只要业务报告保持更新，业务产品线能够持续响应，那么业务流程就可以进行下去，而无需等待特定的响应，也不要求在几毫秒内得到响应。一些业务流程具有持续性和非阻塞的特点，比如针对可疑信用卡交易的警告、网络警告、根据供应关系实时调整价格、跟踪包裹。

流的定义不依赖任何一个特定的框架、API 或特性。只要持续地从一个无边界的数据集读取数据，然后对它们进行处理并生成结果，那就是在进行流式处理。重点是，整个处理过程必须是持续的。一个在每天凌晨两点启动的流程，从流里读取 500 条记录，生成结果，然后结束，这样的流程不是流式处理。

11.2　流式处理的一些概念

流式处理的很多方面与普通的数据处理是很相似的：写一些代码来接收数据，对数据进行处理，可能做一些转换、聚合和增强的操作，然后把生成的结果输出到某个地方。不过流式处理有一些特有的概念，对于那些有数据处理经验但是首次尝试开发流式处理应用程序的人来说，很容易造成混淆。下面将试着澄清这些概念。

11.2.1　时间

时间或许就是流式处理最为重要的概念，也是最让人感到困惑的。在讨论分布式系统时，该如何理解复杂的时间概念？推荐阅读 Justin Sheehy 的论文“There is No Now”。在流式处理里，时间是一个非常重要的概念，因为大部分流式应用的操作都是基于时间窗口的。例如，流式应用可能会计算股价的 5 分钟移动平均数。如果生产者因为网络问题离线了 2 小时，然后带着 2 小时的数据重新连线，我们需要知道该如何处理这些数据。这些数据大部分都已经超过了 5 分钟，而且没有参与之前的计算。

流式处理系统一般包含如下几个时间概念。

事件时间

　　事件时间是指所追踪事件的发生时间和记录的创建时间。例如，度量的获取时间、商店里商品的出售时间、网站用户访问网页的时间，等等。在 Kafka 0.10.0 和更高版本里，生产者会自动在记录中添加记录的创建时间。如果这个时间戳与应用程序对“事件时间”的定义不一样，例如，Kafka 的记录是基于事件发生后的数据库记录创建的，那就需要自己设置这个时间戳字段。在处理数据流时，事件时间是很重要的。

日志追加时间

　　日志追加时间是指事件保存到 broker 的时间。在 Kafka 0.10.0 和更高版本里，如果启用了自动添加时间戳的功能，或者记录是使用旧版本的生产者客户端生成的，而且没有包含时间戳，那么 broker 会在接收这些记录时自动添加时间戳。这个时间戳一般与流式处理没有太大关系，因为用户一般只对事件的发生时间感兴趣。例如，如果要计算每天生产了多少台设备，就需要计算在那一天实际生产的设备数量，尽管这些事件有可能因为网络问题到了第二天才进入 Kafka。不过，如果真实的事件时间没有被记录下来，那么就可以使用日志追加时间，在记录创建之后，这个时间就不会发生改变。

处理时间

处理时间是指应用程序在收到事件之后要对其进行处理的时间。这个时间可以是在事件发生之后的几毫秒、几小时或几天。同一个事件可能会被分配不同的时间戳，这取决于应用程序何时读取这个事件。如果应用程序使用了两个线程来读取同一个事件，这个时间戳也会不一样！所以这个时间戳非常不可靠，应该避免使用它。

> 注意时区问题
>
> 在处理与时间有关的问题时，需要注意时区问题。整个数据管道应该使用同一个时区，否则操作的结果就会出现混淆，变得毫无意义。如果时区问题不可避免，那么在处理事件之前需要将它们转换到同一个时区，这就要求记录里同时包含时区信息。

11.2.2 状态

如果只是单独处理每一个事件，那么流式处理就很简单。例如，如果想从 Kafka 读取在线购物交易事件流，找出金额超过 10 000 美元的交易，并将结果通过邮件发送给销售人员，那么可以使用 Kafka 消费者客户端和 SMTP 库，几行代码就可以搞定。

如果操作里包含了多个事件，流式处理就会变得很有意思，比如根据类型计算事件的数量、移动平均数、合并两个流以便生成更丰富的信息流。在这些情况下，光处理单个事件是不够的，用户需要跟踪更多的信息，比如这个小时内看到的每种类型事件的个数、需要合并的事件、将每种类型的事件值相加，等等。事件与事件之间的信息被称为“状态”。

这些状态一般被保存在应用程序的本地变量里。例如，使用散列表来保存移动计数器。事实上，本书的很多例子就是这么做的。不过，这不是一种可靠的方法，因为如果应用程序关闭，状态就会丢失，结果就会发生变化，而这并不是用户希望看到的。所以，要小心地持久化最近的状态，如果应用程序重启，要将其恢复。

流式处理包含以下几种类型的状态。

本地状态或内部状态

这种状态只能被单个应用程序实例访问，它们一般使用内嵌在应

用程序里的数据库进行维护和管理。本地状态的优势在于它的速度，不足之处在于它受到内存大小的限制。所以，流式处理的很多设计模式都将数据拆分到多个子流，这样就可以使用有限的本地状态来处理它们。

外部状态

　　这种状态使用外部的数据存储来维护，一般使用 NoSQL 系统，比如 Cassandra。使用外部存储的优势在于，它没有大小的限制，而且可以被应用程序的多个实例访问，甚至被不同的应用程序访问。不足之处在于，引入额外的系统会造成更大的延迟和复杂性。大部分流式处理应用尽量避免使用外部存储，或者将信息缓存在本地，减少与外部存储发生交互，以此来降低延迟，而这就引入了如何维护内部和外部状态一致性的问题。

11.2.3　流和表的二元性

大家都熟悉数据库表，表就是记录的集合，每个表都有一个主键，并包含了一系列由 schema 定义的属性。表的记录是可变的（可以在表上面执行更新和删除操作）。我们可以通过查询表数据获知某一时刻的数据状态。例如，通过查询 CUSTOMERS_CONTACTS 这个表，就可以获取所有客户的联系信息。如果表被设计成不包含历史信息，那么就找不到客户过去的联系信息了。

在将表与流进行对比时，可以这么想：流包含了变更——流是一系列事件，每个事件就是一个变更。表包含了当前的状态，是多个变更所产生的结果。所以说，表和流是同一个硬币的两面——世界总是在发生变化，用户有时候关注变更事件，有时候则关注世界的当前状态。如果一个系统允许使用这两种方式来查看数据，那么它就比只支持一种方式的系统强大。

为了将表转化成流，需要捕捉到在表上所发生的变更，将“insert”、“update”和“delete”事件保存到流里。大部分数据库提供了用于捕捉变更的“Change Data Capture”（CDC）解决方案，Kafka 连接器将这些变更发送到 Kafka，用于后续的流式处理。

为了将流转化成表，需要“应用”流里所包含的所有变更，这也叫作流的“物化”。首先在内存里、内部状态存储或外部数据库里创建一个表，然后从头到尾遍历流里的所有事件，逐个地改变状态。在完成这个过程之后，得到了一个表，它代表了某个时间点的状态。

假设有一个鞋店，某零售活动可以使用一个事件流来表示：

“红色、蓝色和绿色鞋子到货”

“蓝色鞋子卖出”

“红色鞋子卖出”

“蓝色鞋子退货”

“绿色鞋子卖出”

如果想知道现在仓库里还有哪些库存，或者到目前为止赚了多少钱，需要对视图进行物化。图 11-1 告诉我们，目前还有蓝色和黄色鞋子，账户上有 170 美元。如果想知道鞋店的繁忙程度，可以查看整个事件流，会发现总共发生了 5 个交易，还可以查出为什么蓝色鞋子被退货。

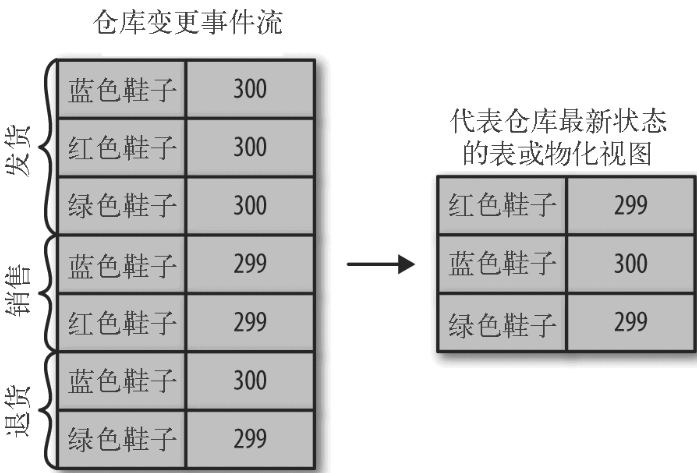


图 **11-1**：物化仓库变更事件流

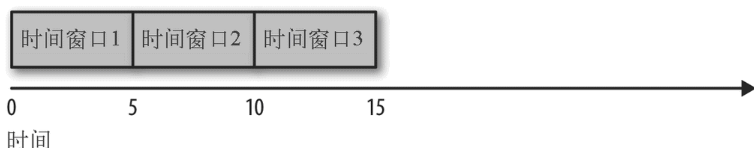
11.2.4　时间窗口

大部分针对流的操作都是基于时间窗口的，比如移动平均数、一周内销量最好的产品、系统的 99 百分位等。两个流的合并操作也是基于时间窗口的，我们会合并发生在相同时间片段上的事件。不过，很少人会停下来仔细想想时间窗口的类型。例如，在计算移动平均数时，需要知道以下几个问题。

- 窗口的大小。是基于 5 分钟进行平均，还是 15 分钟，或者一天？窗口越小，就能越快地发现变更，不过噪声也越多。窗口越大，变更就越平滑，不过延迟也越严重，如果价格涨了，需要更长的时间才能看出来。
- 窗口移动的频率（“移动间隔”）。5 分钟的平均数可以每分钟变化一次，或者每秒钟变化一次，或者每当有新事件到达时发生变化。如果“移动间隔”与窗口大小相等，这种情况被称为“滚动窗口（tumbling window）”。如果窗口随着每一条记录移动，这种情况被称为“滑动窗口（sliding window）”。
- 窗口的可更新时间多长。假设计算了 00:00 到 00:05 之间的移动平均数，一个小时之后又得到了一些“事件时间”是 00:02 的事件，那么需要更新 00:00 到 00:05 这个窗口的结果吗？或者就这么算了？理想情况下，可以定义一个时间段，在这个时间段内，事件可以被添加到与它们相应的时间片段里。如果事件处于 4 个小时以内，那么就更新它们，否则就忽略它们。

窗口可以与时间对齐，比如 5 分钟的窗口如果每分钟移动一次，那么第一个分片可以是 00:00~00:05，第二个就是 00:01~00:06。它也可以不与时间对齐，应用可以在任何时候启动，那么第一个分片有可能是 03:17~03:22。滑动窗口永远不会与时间对齐，因为只要有新记录到达，它们就会发生移动。图 11-2 展示了这两种时间窗口的不同之处。

滚动窗口——5分钟的时间窗口，每5分钟滚动一次



跳跃窗口——5分钟的时间窗口，每分钟跳跃一次
窗口重叠，事件属于多个时间窗口

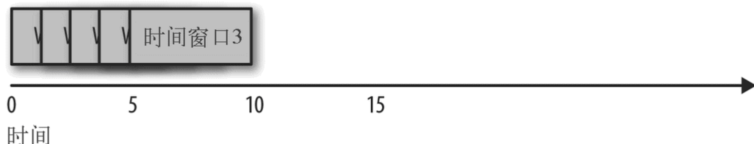


图 11-2：滚动窗口和跳跃窗口的区别

11.3　流式处理的设计模式

每一个流式处理系统都不一样，从基本的消费者、处理逻辑和生产者的组合，到使用了 Spark Streaming 和机器学习软件包的复杂集群，以及其他很多处于中间位置的组件。不过有一些基本的设计模式和解决方案可以满足流式处理架构的常见需求。下面将介绍一些这样的模式，并举例说明如何使用这种模式。

11.3.1　单个事件处理

处理单个事件是流式处理最基本的模式。这个模式也叫 map 或 filter 模式，因为它经常被用于过滤无用的事件或者用于转换事件（map 这个术语是从 Map-Reduce 模式中来的，map 阶段转换事件，reduce 阶段聚合转换过的事件）。

在这种模式下，应用程序读取流中的事件，修改它们，然后把事件生成到另一个流上。比如，一个应用程序从一个流中读取日志消息，并把 ERROR 级别的消息写到高优先级的流中，同时把其他消息写到低优先级的流中。再如，一个应用程序从流中读取事件，并把事件从 JSON 格式改为 Avro 格式。这类应用程序不需要在程序内部维护状态，因为每一个事件都是独立处理的。这也意味着，从错误中恢复或进行负载均衡会非常容易，因为不需要进行恢复状态的操作，只需要将事件交给应用程序的另一个实例去处理。

这种模式可以使用一个生产者和一个消费者来实现，如图 11-3 所示。

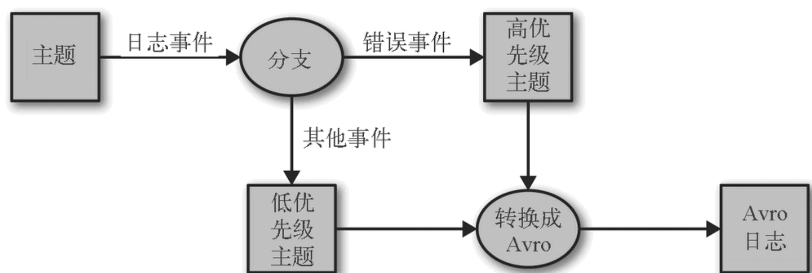


图 **11-3**：单事件处理拓扑

11.3.2　使用本地状态

大部分流式处理应用程序关心的是如何聚合信息，特别是基于时间窗口进行聚合。例如，找出每天最低和最高的股票交易价格并计算移动平均数。

要实现这些聚合操作，需要维护流的状态。在本例中，为了计算每天的最小价格和平均价格，需要将最小值和最大值保存下来，并将它们与每一个新值进行对比。

这些操作可以通过本地状态（而不是共享状态）来实现，因为本例中的每一个操作都是基于组的聚合操作，如图 11-4 所示。例如，基于各个股票代码进行聚合，而不是基于整个股票市场。我们使用了一个 Kafka 分区器来确保具有相同股票代码的事件总是被写入相同的分区。应用程序的每个实例从分配给它们的分区上获取事件（这是 Kafka 的消费者保证）。也就是说，应用程序的每一个实例都可以维护一个股票代码子集的状态。

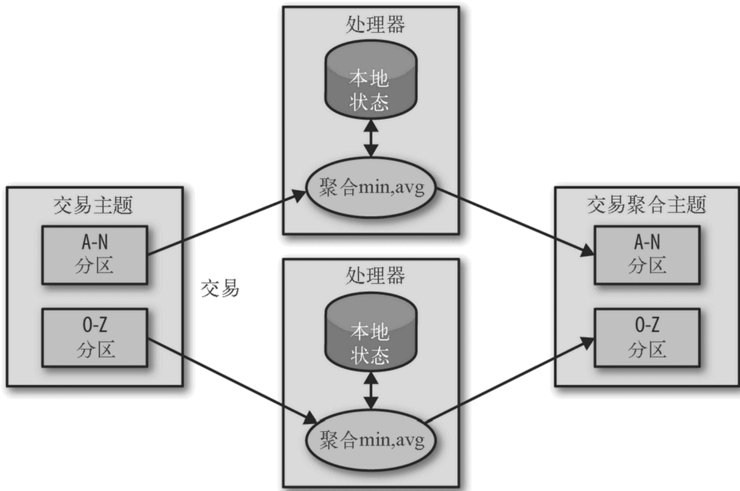


图 **11-4**：使用本地状态的事件拓扑

如果流式处理应用程序包含了本地状态，情况就会变得非常复杂，而且还需要解决下列的一些问题。

内存使用

　　应用实例必须有可用的内存来保存本地状态。

持久化

　　要确保在应用程序关闭时不会丢失状态，并且在应用程序重启后或者切换到另一个应用实例时可以恢复状态。Streams 可以很好地处理这些问题，它使用内嵌的 RocksDB 将本地状态保存在内存里，同

时持久化到磁盘上，以便在重启后可以恢复。本地状态的变更也会被发送到 Kafka 主题上。如果 Streams 节点崩溃，本地状态并不会丢失，可以通过重新读取 Kafka 主题上的事件来重建本地状态。例如，如果本地状态包含“IBM 当前最小价格是 167.19”，并且已经保存到了 Kafka 上，那么稍后就可以通过读取这些数据来重建本地缓存。这些 Kafka 主题使用了压缩日志，以确保它们不会无限量地增长，方便重建状态。

再均衡

　　有时候，分区会被重新分配给不同的消费者。在这种情况下，失去分区的实例必须把最后的状态保存起来，同时获得分区的实例必须知道如何恢复到正确的状态。

不同的流式处理框架为开发者提供了不同的本地状态支持。如果应用程序需要维护本地状态，那么就要知道框架是否提供了支持。本章的末尾将会对一些框架进行简要的对比，不过软件发展变化太快，而流式处理框架更是如此。

11.3.3　多阶段处理和重分区

本地状态对按组聚合操作起到很大的作用。但如果需要使用所有可用的信息来获得一个结果呢？例如，假设要发布每天的“前 10 支”股票，这 10 支股票需要从每天的交易股票中挑选出来。很显然，如果只是在每个应用实例上进行处理是不够的，因为 10 支股票分布在多个实例上，如图 11-5 所示。我们需要一个两阶段解决方案。首先，计算每支股票当天的涨跌，这个可以在每个实例上进行。然后将结果写到一个包含了单个分区的新主题上。另一个单独的应用实例读取这个分区，找出当天的前 10 支股票。新主题只包含了每支股票的概要信息，比其他包含交易信息的主题要小很多，所以流量很小，使用单个应用实例就足以应付。不过，有时候需要更多的步骤才能生成结果。

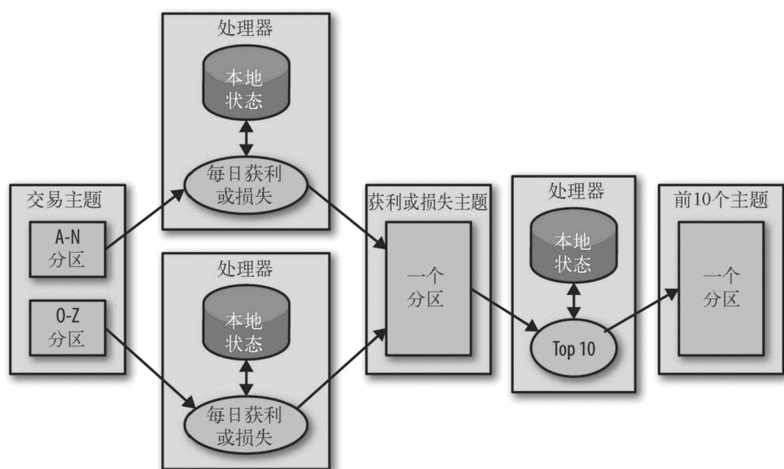


图 **11-5**：包含本地状态和重分区步骤的拓扑

这种多阶段处理对于写过 Map-Reduce 代码的人来说应该很熟悉，因为他们经常要使用多个 reduce 步骤。如果写过 Map-Reduce 代码，就应该知道，处理每个 reduce 步骤的应用需要被隔离开来。与 Map-Reduce 不同的是，大多数流式处理框架可以将多个步骤放在同一个应用里，框架会负责调配每一步需要运行哪一个应用实例（或 worker）。

11.3.4　使用外部查找——流和表的连接

有时候，流式处理需要将外部数据和流集成在一起，比如使用保存在外部数据库里的规则来验证事务，或者将用户信息填充到点击事件当中。

很明显，为了使用外部查找来实现数据填充，可以这样做：对于事件流里的每一个点击事件，从用户信息表里查找相关的用户信息，从中抽取用户的年龄和性别信息，把它们包含在点击事件里，然后将事件发布到另一个主题上，如图 11-6 所示。

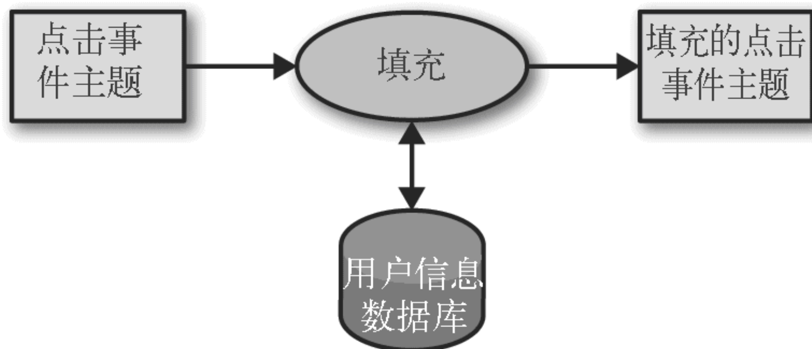


图 **11-6**：使用外部数据源的流式处理

这种方式最大的问题在于，外部查找会带来严重的延迟，一般在 5~15ms 之间。这在很多情况下是不可行的。另外，外部数据存储也无法接受这种额外的负载——流式处理系统每秒钟可以处理 10~50 万个事件，而数据库正常情况下每秒钟只能处理 1 万个事件，所以需要伸缩性更强的解决方案。

为了获得更好的性能和更强的伸缩性，需要将数据库的信息缓存到流式处理应用程序里。不过，要管理好这个缓存也是一个挑战。比如，如何保证缓存里的数据是最新的？如果刷新太频繁，那么仍然会对数据库造成压力，缓存也就失去了作用。如果刷新不及时，那么流式处理中所用的数据就会过时。

如果能够捕捉数据库的变更事件，并形成事件流，流式处理作业就可以监听事件流，并及时更新缓存。捕捉数据库的变更事件并形成事件流，这个过程被称为 CDC——变更数据捕捉（Change Data Capture）。如果使用了 Connect，就会发现，有一些连接器可以用于执行 CDC 任务，把数据库表转成变更事件流。这样就拥有了数据库表的私有副本，一旦数据库发生变更，用户会收到通知，并根据变更事件更新私有副本里的数据，如图 11-7 所示。

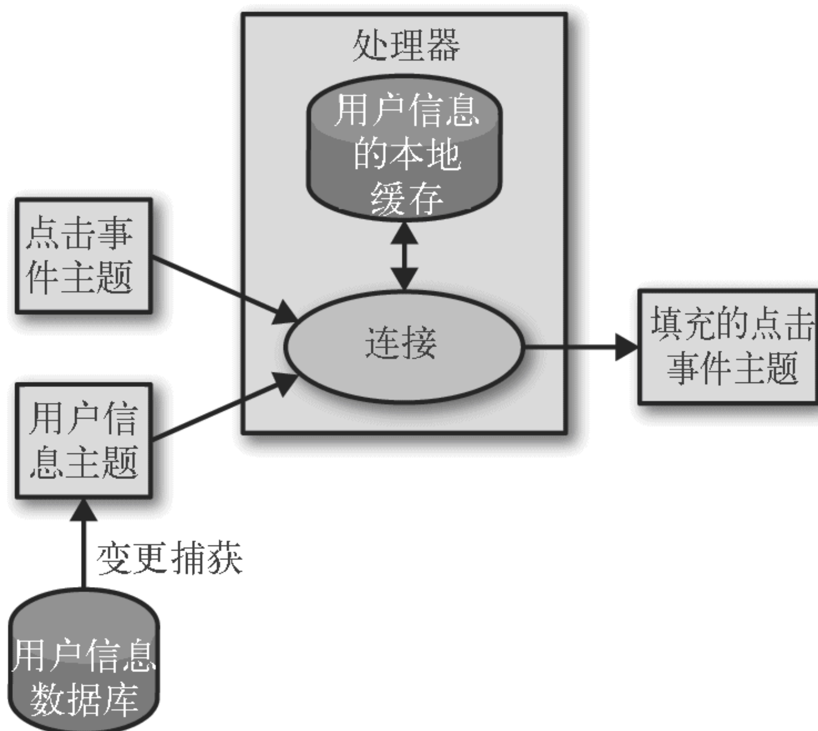


图 **11-7**：连接流和表的拓扑，不需要外部数据源

这样一来，当收到点击事件时，可以从本地的缓存里查找 user_id，并将其填充到点击事件里。因为使用的是本地缓存，它具有更强的伸缩性，而且不会影响数据库和其他使用数据库的应用程序。

之所以将这种方案叫作流和表的连接，是因为其中的一个流代表了本地缓存表的变更。

11.3.5　流与流的连接

有时候需要连接两个真实的事件流。什么是“真实”的流？本章开始的时候曾经说过，流是无边界的。如果使用一个流来表示一个表，那么就可以忽略流的大部分历史事件，因为你只关心表的当前状态。不过，如果要连接两个流，那么就是在连接所有的历史事件——将两个流里具有相同键和发生在相同时间窗口内的事件匹配起来。这就是为什么流和流的连接也叫作基于时间窗口的连接（windowed-join）。

假设有一个由网站用户输入的搜索事件流和一个由用户对搜索结果进

行点击的事件流。对用户的搜索和用户对搜索结果的点击进行匹配，就可以知道哪一个搜索的热度更高。很显然，我们需要基于搜索关键词进行匹配，而且每个关键词只能与一定时间窗口内的事件进行匹配——假设用户在输入搜索关键词后几秒钟就会点击搜索结果。因此，我们为每一个流维护了以几秒钟为单位的时间窗口，并对这些时间窗口事件结果进行匹配，如图 11-8 所示。

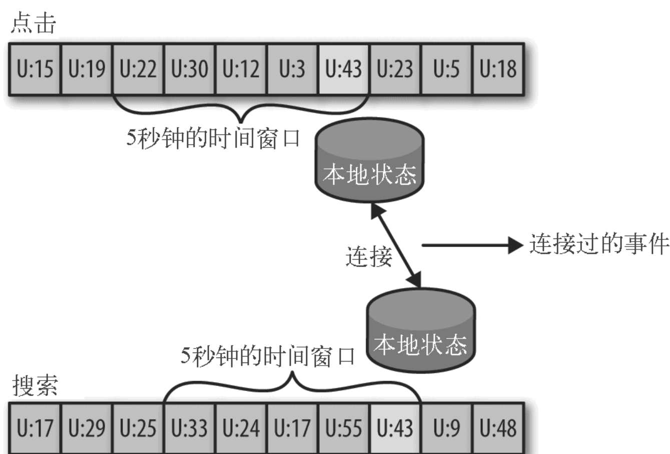


图 **11-8**：连接两个流，通常包含一个移动时间窗

在 Streams 中，上述的两个流都是通过相同的键来进行分区的，这个键也是用于连接两个流的键。这样一来，user_id:42 的点击事件就被保存在点击主题的分区 5 上，而所有 user_id:42 的搜索事件被保存在搜索主题的分区 5 上。Streams 可以确保这两个主题的分区 5 的事件被分配给同一个任务，这个任务就会得到所有与 user_id:42 相关的事件。Streams 在内嵌的 RocksDB 里维护了两个主题的连接时间窗口，所以能够执行连接操作。

11.3.6　乱序的事件

不管是对于流式处理还是传统的 ETL 系统来说，处理乱序事件都是一个挑战。物联网领域经常发生乱序事件：一个移动设备断开 WiFi 连接几个小时，在重新连上 WiFi 之后将几个小时累积的事件一起发送出去，如图 11-9 所示。这在监控网络设备（故障交换机被修复之前不会发送任何诊断数据）或进行生产（装置间的网络连接非常不可靠）时也时有发生。



图 **11-9**：乱序事件

要让流处理应用程序处理好这些场景，需要做到以下几点。

- 识别乱序的事件。应用程序需要检查事件的时间，并将其与当前时间进行比较。
- 规定一个时间段用于重排乱序的事件。比如 3 个小时以内的事件可以重排，但 3 周以外的事件就可以直接扔掉。
- 具有在一定时间段内重排乱序事件的能力。这是流式处理应用与批处理作业的一个主要不同点。假设有一个每天运行的作业，一些事件在作业结束之后才到达，那么可以重新运行昨天的作业来更新事件。而在流式处理中，“重新运行昨天的作业”这种情况是不存在的，乱序事件和新到达的事件必须一起处理。
- 具备更新结果的能力。如果处理的结果保存到数据库里，那么可以通过 put 或 update 对结果进行更新。如果流应用程序通过邮件发送结果，那么要对结果进行更新，就需要很巧妙的手段。

有一些流式处理框架，比如 Google 的 Dataflow 和 Kafka 的 Streams，都支持独立于处理时间发生的事件，并且能够处理比当前处理时间更晚或更早的事件。它们在本地状态里维护了多个聚合时间窗口，用于更新事件，并为开发者提供配置时间窗口大小的能力。当然，时间窗口越大，维护本地状态需要的内存也越大。

Streams API 通常将聚合结果写到主题上。这些主题一般是压缩日志主题，也就是说，它们只保留每个键的最新值。如果一个聚合时间窗口的结果需要被更新为晚到事件的结果， Streams 会直接为这个聚合时间窗口写入一个新的结果，将前一个结果覆盖掉。

11.3.7　重新处理

最后一个很重要的模式是重新处理事件，该模式有两个变种。

- 我们对流式处理应用进行了改进，使用新版本应用处理同一个

事件流，生成新的结果，并比较两种版本的结果，然后在某个时间点将客户端切换到新的结果流上。
- 现有的流式处理应用出现了缺陷，修复缺陷之后，重新处理事件流并重新计算结果。

对于第一种情况，Kafka 将事件流长时间地保存在可伸缩的数据存储里。也就是说，要使用两个版本的流式处理应用来生成结果，只需要满足如下条件：

- 将新版本的应用作为一个新的消费者群组；
- 让它从输入主题的第一个偏移量开始读取数据（这样它就拥有了属于自己的输入流事件副本）；
- 检查结果流，在新版本的处理作业赶上进度时，将客户端应用程序切换到新的结果流上。

第二种情况有一定的挑战性。它要求“重置”应用，让应用回到输入流的起始位置开始处理，同时重置本地状态（这样就不会将两个版本应用的处理结果混淆起来了），而且还可能需要清理之前的输出流。虽然 Streams 提供了一个工具用于重置应用的状态，不过如果有条件运行两个应用程序并生成两个结果流，还是建议使用第一种方案。第一种方案更加安全，多个版本可以来回切换，可以比较不同版本的结果，而且不会造成数据的丢失，也不会在清理过程中引入错误。

11.4　Streams示例

为了演示如何在实际中实现这些模式，下面将给出一些使用 Streams API 的例子。之所以使用这个 API，是因为它相对简单，而且它是与 Kafka 一起发布的，用户可以直接使用它。不过要记住一点，我们可以使用任意的流式处理框架和软件包来实现这些模式，这些模式具有通用性。

Kafka 有两个基于流的 API，一个是底层的 Processor API，一个是高级的 Streams DSL。下面的例子中将使用 Streams DSL。通过为事件流定义转换链可以实现流式处理。转换可以是简单的过滤器，也可以是复杂的流与流的连接。我们可以通过底层的 API 实现自己的转换，不过没必要这么做。

在使用 DSL API 时，一般会先用 StreamBuilder 创建一个拓扑（topology）。拓扑是一个有向图（DAG），包含了各个转换过程，将会被应用在流的事件上。在创建好拓扑后，使用拓扑创建一个

KafkaStreams 执行对象。多个线程会随着 KafkaStreams 对象启动，将拓扑应用到流的事件上。在关闭 KafkaStreams 对象时，处理也随之结束。

下面将展示一些使用 Streams 来实现上述模式的例子。字数统计这个例子用于演示 map 与 filter 模式以及简单的聚合，另一个例子是计算股票交易市场的各种统计信息，用于演示基于时间窗口的聚合，最后使用填充点击事件流（ClickStream Enrichment）的例子来演示流的连接。

11.4.1　字数统计

先看一个使用了 Streams 的字数统计示例。完整的示例代码可以在 GitHub（https://github.com/gwenshap/kafka-streams-wordcount）上找到。

要创建一个流式处理应用，首先需要配置 Kafka Streams 引擎。KafkaStreams 有很多配置参数，这里就不展开讨论了，感兴趣的读者可以在官方文档里查看。另外，也可以将生产者和消费者内嵌到 Kafka Streams 引擎里，只要把生产者或消费者的配置信息添加到 Properties 对象里即可。

```
public class WordCountExample {

    public static void main(String[] args) throws Exception{

        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG,
          "wordcount"); ❶
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
          "localhost:9092"); ❷
        props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
          Serdes.String().getClass().getName()); ❸
        props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
          Serdes.String().getClass().getName());
```

❶ 每个 Streams 应用程序必须要有一个应用 ID。这个 ID 用于协调应用实例，也用于命名内部的本地存储和相关主题。对于同一个 Kafka 集群里的每一个 Streams 应用来说，这个名字必须是唯一的。

❷ Streams 应用程序从 Kafka 主题上读取数据，并将结果写到 Kafka 主题上，所以我们要告诉应用程序如何找到 Kafka。稍后我们将介绍，Streams 应用程序也使用 Kafka 作为协调工具。

❸ 在读写数据时，应用程序需要对消息进行序列化和反序列化，因此提供了默认的序列化类和反序列化类。如果有必要，可以在稍后创建拓扑时覆盖默认的类。

做好配置之后，下面开始创建拓扑。

```
KStreamBuilder builder = new KStreamBuilder(); ❶

KStream<String, String> source =
  builder.stream("wordcount-input");


final Pattern pattern = Pattern.compile("\\W+");

KStream counts  = source.flatMapValues(value->
  Arrays.asList(pattern.split(value.toLowerCase()))) ❷
        .map((key, value) -> new KeyValue<Object,
          Object>(value, value))
        .filter((key, value) -> (!value.equals("the"))) ❸
        .groupByKey() ❹
        .count("CountStore").mapValues(value->
          Long.toString(value)).toStream(); ❺
counts.to("wordcount-output"); ❻
```

❶ 创建一个 `KStreamBuilder` 对象，并定义了一个流，将它指向输入主题。

❷ 从主题上读取的每一个事件就是一行文字，首先使用正则表达式将它拆分为一系列单词，然后将每个单词（事件的值）作为事件的键，这样就可以执行 group by 操作了。

❸ 将单词 the 过滤掉，过滤操作看起来很简单。

❹ 根据键执行 group by 操作，这样就得到了一个不重复的单词集合。

❺ 计算每个集合里的事件数。计算的结果是 Long 类型，将它转成 String 类型，方便从 Kafka 上读取结果。

❻ 最后把结果写回 Kafka。

定义好转换的流程后，应用程序将会运行这个流程，接下来要做的就是运行它。

```
        KafkaStreams streams = new KafkaStreams(builder, props); ❶
```

```
        streams.start(); ❷

        // 一般情况下，Streams应用程序会一直运行下去
        // 本例中，只让它运行一段时间，然后停掉它，因为数据是有限的
        Thread.sleep(5000L);

        streams.close(); ❸
    }
}
```

❶ 基于拓扑和配置属性定义一个 `KafkaStreams` 对象。

❷ 启动 Kafka Streams 引擎。

❸ 过一段时间后将它停掉。

就是这么简单！本例只用了几行代码，就演示了如何实现单事件处理模式（在事件上使用了 map 与 filter），然后通过 group by 操作对数据进行重新分区，并为统计记录个数维护了一个简单的本地状态。

建议运行完整的示例，GitHub 库的 README 文件包含了如何运行示例的说明。

也许你会注意到，除了 Kafka 外，不需要在机器上安装任何软件，就可以运行完整的示例。这类似于在“本地模式”下使用 Spark。主要的不同之处在于，如果主题包含了多个分区，就可以运行多个字数统计应用实例（在不同的命令行终端运行），而这也就是第一个 Streams 集群。几个字数统计应用实例之间互相交互，协调处理任务。Spark 的本地模式非常简单，但要在生产环境运行集群，需要安装 YARN 或者 Mesos，并在所有的机器上安装 Spark，然后将你的应用提交到集群上，所以 Spark 有较高的准入门槛。而如果使用 Streams API，只需要启动几个应用实例就可以拥有一个集群。在开发机上运行和在生产环境中运行几乎是一样的。

11.4.2　股票市场统计

接下来的这个例子会复杂一些，下面将从一个股票交易事件流里读取事件，这些事件包含了股票代码、沽盘价和要价规模。在股票交易里，沽盘价是指卖方的出价，买入价是指买方建议支付的价格，要价规模是指卖方愿意在相应价格基础上出售的股数。为了简单起见，这里直接忽略竞标过程。数据里不会包含时间戳，相反，我们会使用由

Kafka 生产者计算得出的“事件时间”。

下面将创建一个包含了若干时间窗口统计信息的输出流：

- 每 5 秒钟内最好的（最低的）沽盘价；
- 每 5 秒钟内交易的股数；
- 每 5 秒钟内平均沽盘价。

这些统计信息每秒钟会更新一次。

为了简单起见，假设交易所只有 10 支不同的股票。应用的参数设置与字数统计示例很相似。

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "stockstat");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
Constants.BROKER);
props.put(StreamsConfig.KEY_SERDE_CLASS_CONFIG,
Serdes.String().getClass().getName());
props.put(StreamsConfig.VALUE_SERDE_CLASS_CONFIG,
TradeSerde.class.getName());
```

主要的不同在于这次使用的 `Serde` 类是不一样的。在字数统计应用里，键和值的类型都是 String，所以使用了 `Serdes.String()` 类作为序列化器和反序列化器。而在这个例子里，键仍然是一个字符串，但值是一个 `Trade` 对象，它包含了股票代码、沽盘价和要价规模。为了序列化和反序列化这个对象（还包括应用里用到的其他几种对象），使用了 Google 的 Gson 类库。借助这个类库，可以生成 JSon 序列化器和反序列化器，然后创建一个包装类，用于生成 `Serde` 对象。

```
static public final class TradeSerde extends WrapperSerde<Trade> {
    public TradeSerde() {
        super(new JsonSerializer<Trade>(),
          new JsonDeserializer<Trade>(Trade.class));
    }
}
```

这里没有什么蹊跷的，只是要记得为存储在 Kafka 里的每一个对象提供一个 `Serde` 对象——输入、输出和中间结果。为了简化这个过程，建议使用 GSon、Avro、Protobuf 等框架来生成 Serde。

在配置好以后，下面开始构建拓扑。

```
KStream<TickerWindow, TradeStats> stats = source.groupByKey() ❶
        .aggregate(TradeStats::new, ❷
            (k, v, tradestats) -> tradestats.add(v), ❸
            TimeWindows.of(5000).advanceBy(1000), ❹
            new TradeStatsSerde(), ❺
            "trade-stats-store") ❻
        .toStream((key, value) -> new TickerWindow(key.key(),
          key.window().start())) ❼
        .mapValues((trade) -> trade.computeAvgPrice()); ❽

        stats.to(new TickerWindowSerde(), new TradeStatsSerde(),
          "stockstatsoutput"); ❾
```

❶ 本例从输入主题上读取事件并执行一个 groupByKey() 操作开始。这个方法虚有其名，实际上并不会执行任何分组操作。不过，它会确保事件流按照记录的键进行分区。因为在写数据时使用了键，而且在调用 groupByKey() 方法之前不会对键进行修改，数据仍然是按照它们的键进行分区的，所以说这个方法不会做任何事情。

❷ 在确保正确的分区之后，开始进行基于时间窗口的聚合。aggregate 方法将流拆分成相互叠加的时间窗口（每秒钟出现一个 5 秒钟的时间窗口），然后在时间窗口内的所有事件上应用聚合方法。这个方法的第一个参数是一个新的对象，用于存放聚合的结果，也就是 Tradestats。我们创建这个对象，并用它存放每个时间窗口的统计信息——最低价格、平均价格和交易数量。

❸ 提供了一个方法对记录进行聚合，Tradestats 的 add 方法用于更新窗口内的最低价格、交易数量和总价。

❹ 定义了 5s（5000ms）的时间窗口，并且每秒钟都会向前滑动。

❺ 提供了一个 Serde 对象，用于序列化和反序列化聚合结果（Tradestats 对象）。

❻ 在介绍模式时曾经说过，基于时间窗口的聚合需要维护本地状态。聚合方法的最后一个参数就是本地状态存储的名字，它可以是任意具有唯一性的名字。

❼ 聚合结果是一个表，包含了股票信息，并使用时间窗口作为主键、聚合结果作为值。它表示一条记录，以及从变更流中计算得出的特定状态（参考“概念”一节有关流和表二元性的讨论）。这里想将表重新

转成事件流，不过不再使用整个时间窗口作为键，而是使用一个包含了股票信息和时间窗口起始时间的对象。toStream 方法将表转成一个流，并将键转成 TickerWindow 对象。

❽ 最后一步是更新平均价格。现在，聚合结果里包含了总价和交易数量。遍历所有的记录，并使用现有的统计信息计算平均价格，然后把它写到输出流里。

❾ 最后将结果写到 stockstats-output 流里。

定义好流程之后，用它生成 KafkaStreams 对象，并运行它，就像之前的“字数统计”那个例子一样。

这个例子展示了如何在一个流上面进行基于时间窗的聚合，这也许就是最为流行的流式处理使用场景。大家可以发现，为聚合维护一个本地状态是一件多么简单的事情，只需要提供一个 Serde 对象和状态存储的名字。不仅如此，这个应用还能扩展到多个实例，如果有实例失效，它的分区将会被分配给其他存活的实例，从而实现自动的故障恢复。在 11.5 节将介绍更多有关这种机制的实现原理。

完整的例子和运行说明可以在 GitHub（https://github.com/gwenshap/kafka-streams-stockstats）上找到。

11.4.3　填充点击事件流

最后一个例子将通过填充网站点击事件流来演示如何进行流的连接。本例首先生成一个模拟点击的事件流，一个虚拟用户信息数据库表的更新事件流和一个网站搜索事件流，然后将这 3 个流连接起来，从而得到用户活动的 360°视图，比如用户每分钟的搜索内容、点击的内容以及感兴趣的内容。这些连接操作为数据分析提供了丰富的数据集。产品推荐一般就是基于这些信息进行的，比如用户搜索了自行车，点击了“Trek”的链接，并且爱好旅游，那么就可以向用户推荐 Trek 自行车、头盔和具有异国情调的自行车骑行活动（比如去内布拉斯加州）。

应用的配置与前一个例子很相似，所以这里跳过这一步，直接进入到构建连接流要使用到的拓扑。

```
KStream<Integer, PageView> views =
builder.stream(Serdes.Integer(),
new PageViewSerde(), Constants.PAGE_VIEW_TOPIC); ❶
KStream<Integer, Search> searches =
```

```
builder.stream(Serdes.Integer(),new SearchSerde(),
Constants.SEARCH_TOPIC);
KTable<Integer, UserProfile> profiles =
builder.table(Serdes.Integer(), new Pro-fileSerde(),
Constants.USER_PROFILE_TOPIC, "profile-store"); ❷

KStream<Integer, UserActivity> viewsWithProfile = views.leftJoin(profiles,
    (page, profile) -> new UserActivity(profile.getUserID(),
    profile.getUserName(), profile.getZipcode(),
    profile.getInterests(), "", page.getPage())); ❹

KStream<Integer, UserActivity> userActivityKStream =
viewsWithProfile.leftJoin(searches, ❺
    (userActivity, search) ->
    userActivity.updateSearch(search.getSearchTerms()), ❻
    JoinWindows.of(1000), Serdes.Integer(),
    new UserActivitySerde(), new SearchSerde()); ❼
```

❶ 首先为点击事件和搜索事件创建流对象。

❷ 为用户信息定义一个 KTable。KTable 是本地缓存，可以通过变更流来对其进行更新。

❸ 将点击事件流与信息表连接起来，将用户信息填充到点击事件里。在一个流和表的连接操作里，每个事件都会收到来自信息表缓存副本里的信息。这是一个左连接操作，所以如果有的点击事件没有匹配的用户信息，这些事件仍然会被保留下来。

❹ 这是连接方法，它接受两个参数，一个来自事件流，一个来自表记录，并返回一个值。如果是在数据库里，必须知道如何将两个值合并成一个结果。这里创建了一个 `activity` 对象，它包含了用户的详细信息和用户浏览过的页面。

❺ 接下来要将点击信息和用户的搜索事件连接起来。这也是一个左连接操作，不过现在连接的是两个流，而不是流和表。

❻ 这是连接方法，这里只是简单地将搜索关键词添加到匹配的页面视图。

❼ 这是最有意思的部分，流和流之间的连接是基于相同的时间窗口。如果只是把每个用户所有的点击事件和所有的搜索事件连接起来，并没有什么意义，我们要把具有相关性的搜索事件和点击事件连接起来。具有相关性的点击事件应该发生在搜索之后的一小段时间内。所以这里定义了一个一秒钟的连接时间窗口。在搜索之后的一秒钟内发

生的点击事件才被认为是具有相关性的，而且搜索关键词也会被放进包含了点击信息和用户信息的活动记录里，这样有助于对搜索和搜索结果进行全面的分析。

定义了流程之后，用它生成 `KafkaStreams` 对象，并运行它，就像之前的“字数统计”那个例子一样。

这个例子演示了两种不同类型的连接模式，一个是连接流和表，用于将表里的信息填充到流的事件里。这个与在数据仓库里运行查询时加入一个维度的事实表有点相似。第二个是连接基于时间窗口的两个流。这种操作只会在流式处理中出现。

完整的例子和运行说明可以在 GitHub（https://github.com/gwenshap/kafka-clickstream-enrich/tree/master）上找到。

11.5　Kafka Streams的架构概览

11.4.3 节的例子演示了如何使用 Streams API 实现流式处理的设计模式。为了更好地理解 Streams 的工作原理和它的伸缩性，下面深入了解 API 背后的设计原则。

11.5.1　构建拓扑

每个流式应用程序至少会实现和执行一个拓扑。拓扑（在其他流式处理框架里叫作 DAG，即有向无环图）是一个操作和变换的集合，每个事件从输入到输出都会流经它。在之前的字数统计示例里，拓扑结构如图 11-10 所示。

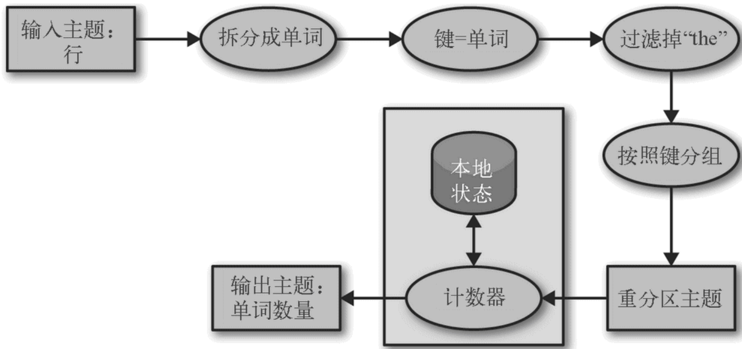


图 **11-10**：字数统计示例的拓扑结构

哪怕是一个很简单的应用，都需要一个拓扑。拓扑是由处理器组成的，这些处理器是拓扑图里的节点（用椭圆表示）。大部分处理器都实现了一个数据操作——过滤、映射、聚合等。数据源处理器从主题上读取数据，并传给其他组件，而数据池处理器从上一个处理器接收数据，并将它们生成到主题上。拓扑总是从一个或多个数据源处理器开始，并以一个或多个数据池处理器结束。

11.5.2　对拓扑进行伸缩

Streams 通过在单个实例里运行多个线程和在分布式应用实例间进行负载均衡来实现伸缩。用户可以在一台机器上运行 Streams 应用，并开启多个线程，也可以在多台机器上运行 Streams 应用。不管采用何种方式，所有的活动线程将会均衡地处理工作负载。

Streams 引擎将拓扑拆分成多个子任务来并行执行。拆分成多少个任务取决于 Streams 引擎，同时也取决于主题的分区数量。每个任务负责一些分区：任务会订阅这些分区，并从分区读取事件数据，在将结果写到数据池之前，在每个事件上执行所有的处理步骤。这些任务是 Streams 引擎最基本的并行单元，因为每个任务可以彼此独立地执行，如图 11-11 所示。

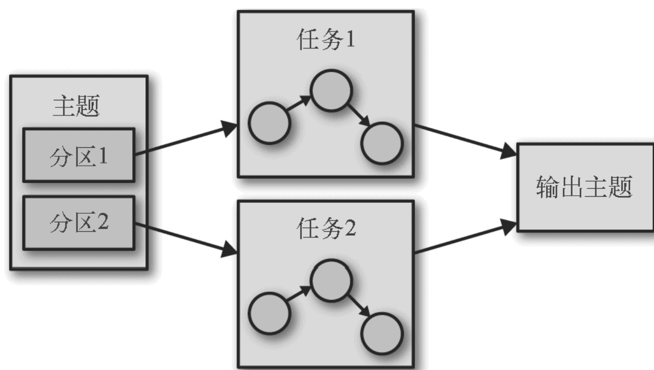


图 **11-11**：运行相同拓扑的两个任务——每个读取主题的一个分区

开发人员可以选择每个应用程序使用的线程数。如果使用了多个线程，每个线程将会执行一部分任务。如果有多个应用实例运行在多个服务器上，每个服务器上的每一个线程都会执行不同的任务。这就是流式应用的伸缩方式：主题里有多少分区，就会有多少任务。如果想要处理得更快，就添加更多的线程。如果一台服务器的资源被用光了，就在另一台服务器上启动应用实例。Kafka 会自动地协调工作，

它为每个任务分配属于它们的分区，每个任务独自处理自己的分区，并维护与聚合相关的本地状态，如图 11-12 所示。

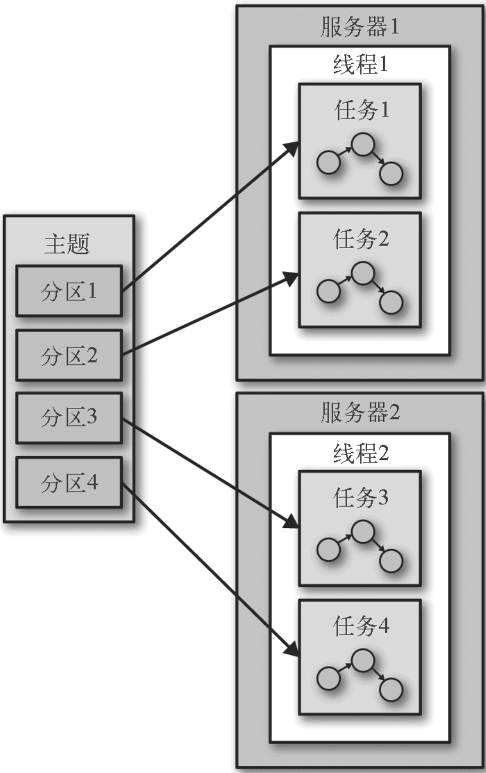


图 **11-12**：处理任务可以运行在多个线程和多个服务器上

大家或许已经注意到，有时候一个步骤需要处理来自多个分区的结果，这样就会在任务之间形成依赖。例如，在点击事件流的例子里对两个流进行了连接，在生成结果之前，需要从每一个流的分区里获取数据。Streams 将连接操作所涉及的分区全部分配给相同的任务，这样，这个任务就可以从相关的分区读取数据，并独立执行连接操作。这也就是为什么 Streams 要求同一个连接操作所涉及的主题必须要有相同数目的分区，而且要基于连接所使用的键进行分区。

如果应用程序需要进行重新分区，也会在任务之间形成依赖。例如，在点击事件流的例子里，所有的事件使用用户 ID 作为键。如果想要基于页面或者邮政编码生成统计信息该怎么办？此时就需要使用邮政编码对数据进行重新分区，并在新分区上运行聚合操作。如果任务 1

处理来自分区 1 的数据，这些数据到达另一个处理器，这个处理器对数据进行重新分区（groupBy 操作），它需要对数据进行 **shuffle**，也就是把数据发送给其他任务进行处理。与其他流式处理框架不一样的是，Streams 通过使用新的键和分区将事件写到新的主题来实现重新分区，并启动新的任务从新主题上读取和处理事件。重新分区的步骤是将拓扑拆分成两个子拓扑，每个子拓扑都有自己的任务集，如图 11-13 所示。第二个任务集依赖第一个任务集，因为它们处理的是第一个子拓扑的结果。不过，它们仍然可以独立地并行执行，因为第一个任务集以自己的速率将数据写到一个主题上，而第二个任务集也以自己的速率从这个主题读取和处理事件。两个任务集之间不需要通信，也没有共享资源，而且它们也不需要运行在相同的线程里或相同的服务器上。这是 Kafka 提供的最有用的特性之一——减少管道各个部分之间的依赖。

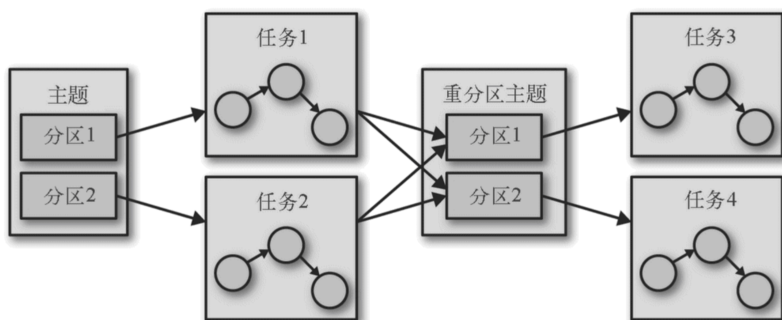


图 **11-13**：处理主题分区事件的两组任务

11.5.3　从故障中存活下来

Streams 的伸缩模型不仅允许伸缩应用，还能优雅地处理故障。首先，包括本地状态在内的所有数据被保存到有高可用性的 Kafka 上。如果应用程序出现故障需要重启，可以从 Kafka 上找到上一次处理的数据在流中的位置，并从这个位置开始继续处理。如果本地状态丢失（比如可能需要将服务器替换掉），应用程序可以从保存在 Kafka 上的变更日志重新创建本地状态。

Streams 还利用了消费者的协调机制来实现任务的高可用性。如果一个任务失败，只要还有其他线程或者应用程序实例可用，就可以使用另一个线程来重启该任务。这类似于消费者群组的故障处理，如果一个消费者失效，就把分区分配给其他活跃的消费者。

11.6　流式处理使用场景

前面已经讲解了如何进行流式处理——从一般性的概念和模式说起，并列举了一些 Streams 的例子。现在是时候让大家知道流式处理都有哪些常见的使用场景了。本章的开头解释过，如果想快速处理事件，而不是为每个批次等上几个小时，但又不是真的要求毫秒级的响应，那么流式处理（或者说持续处理）就可以派上用场了。话是没错，不过听起来仍然十分抽象。下面来看一些例子，它们都使用流式处理来解决实际的问题。

客户服务

　　假设你向一个大型的连锁酒店预订了一个房间，并希望收到邮件确认和票据。在预订了几分钟之后，仍然没有收到确认邮件，于是打电话向客服确认。客服的回复是：“我在我们的系统里看不到订单，不过从预订系统加载数据的批次作业每天只运行一次，所以请明天再打电话过来。你应该可以在 2～3 个工作日之后收到确认邮件。”这样的服务有点糟糕，不过有人已经不止一次地在一家大型连锁酒店遭遇过类似的问题。我们真正需要的是，连锁酒店的每一个系统在预订结束之后的几秒钟或者几分钟之内都能发出通知，包括客服中心、酒店、发送确认邮件的系统、网站等。有的用户可能还希望客服中心能够立即获知自己在这家连锁酒店的历史入住数据，前台能够知道他是一个忠实的客户，从而提供更高级别的服务。如果使用流式处理应用来构建这些系统，就可以实现几近实时的接收和处理这些事件，从而带来更好的用户体验。如果有这样的系统，就可以在几分钟之内收到邮件确认，信用卡就可以及时扣款，然后发送票据，服务台就可以马上回答有关预订房间的问题了。

物联网

　　物联网包含很多东西，从用于调节温度和自动添加洗衣剂的家居设备，到制药行业的实时质量监控设备。流式处理在传感器和设备上应用，最为常见的是用于预测何时该进行设备维护。这个与应用监控有点相似，不过这次是应用在硬件上，而且应用在很多不同的行业——制造业、通信（识别故障基站）、有线电视（在用户投诉之前识别出故障机顶盒）等。每一种场景都有自己的特点，不过目标是一样的——处理大量来自设备的事件，并识别出一些模式，这些模式预示着某些设备需要进行维护，比如交换机数据包的下降、生产过程中需要更大的力气来拧紧螺丝，或者用户频繁重启有线电视的机顶盒。

- 欺诈检测。欺诈检测也被叫作异常检查，是一个非常广泛的领域，专注于捕捉系统中的“作弊者”或不良分子，比如信用卡欺诈、股票交易欺诈、视频游戏作弊或者网络安全风险。在这些欺诈行为造成大规模的破坏之前，越早将它们识别出来越好。一个几近实时的系统可以快速地对事件作出响应，停止一个还没有通过审核的交易要比等待批次作业在 3 天之后才发现它是一个欺诈交易要更容易处理。这也是一个在大规模事件流里识别模式的问题。

  在网络安全领域，有一个被称为发信标（beaconing）的欺诈手法，黑客在组织内部放置恶意软件，该软件时不时地连接到外部网络接收命令。一般来说，网络可以抵挡来自外部的攻击，但难以阻止内部到外部的突围。通过处理大量的网络连接事件流，识别出不正常的通信模式，检测出该主机不经常访问的某些 IP 地址，在蒙受更大的损失之前向安全组织发出告警。

11.7 如何选择流式处理框架

在比较两个流式处理系统时，要着重考虑使用场景是什么。以下是一些需要考虑的应用类别。

摄取

摄取的目的是将数据从一个系统移动到另一个系统，并在传输过程中对数据进行一些修改，使其更适用于目标系统。

低延迟

任何要求立即得到响应的应用。有些欺诈检测场景就属于这一类。

异步微服务

这些微服务为大型的业务流程执行一些简单操作，比如更新仓储信息。这些应用需要通过维护本地状态缓存来提升性能。

几近实时的数据分析

这些流式媒体应用程序执行复杂的聚合和连接，以便对数据进行切分，并生成有趣的业务见解。

选择何种流式处理系统取决于要解决什么问题。

- 如果要解决摄取问题，那么需要考虑一下是需要一个流式处理系统还是一个更简单的专注于摄取的系统，比如 Kafka Connect。如果确定需要一个流式处理系统，那就要确保它拥有可用的连接器，并且要保证目标系统也有高质量的连接器可用。
- 如果要解决的问题要求毫秒级的延迟，那么就要考虑一下是否一定要用流。一般来说，请求与响应模式更加适用于这种任务。如果确定需要一个流式处理系统，那就需要选择一个支持低延迟的模型，而不是基于微批次的模型。
- 如果要构建异步微服务，那么需要一个可以很好地与消息总线（希望是 Kafka）集成的流式处理系统。它应该具备变更捕捉能力，这样就可以将上游的变更传递到微服务本地的缓存里，而且它要支持本地存储，可以作为微服务数据的缓存和物化视图。
- 如果要构建复杂的数据分析引擎，那么也需要一个支持本地存储的流式处理系统，不过这次不是为了本地缓存和物化视图，而是为了支持高级的聚合、时间窗口和连接，因为如果没有本地存储，就很难实现这些特性。API 需要支持自定义聚合、基于时间窗口的操作和多类型连接。

除了使用场景外，还有如下一些全局的考虑点。

系统的可操作性

　　它是否容易部署？是否容易监控和调试？是否易于伸缩？它是否能够很好地与已有的基础设施集成起来？如果出现错误，需要重新处理数据，这个时候该怎么办？

API 的可用性和调试的简单性

　　为了开发出高质量的应用，同一种框架的不同版本可能需要耗费不同的时间，这类情况很常见。开发时间和上市时机太重要了，所以我们需要选择一个高效率的系统。

让复杂的事情简单化

　　几乎每一个系统都声称它们支持基于时间窗口的高级聚合操作和本地缓存，但问题是，它们够简单吗？它们是处理了规模伸缩和故障恢复方面的细节问题，还是只提供了脆弱的抽象，然后让你来处理剩

下的事情？系统提供的 API 越简洁，封装的细节越多，开发人员的效率就越高。

社区

　　大部分流式处理框架都是开源的。对于开源软件来说，一个充满生气的社区是不可替代的。好的社区意味着用户可以定期获得新的功能特性，而且质量相对较高（没有人会使用糟糕的软件），缺陷可以很快地得到修复，而且用户的问题可以及时得到解答。这也意味着，如果遇到一个奇怪的问题并在 Google 上搜索，可以搜索到相关的信息，因为其他人也在使用这个系统，而且也遇到了相同的问题。

11.8　总结

本章的开头解释了流式处理，给出了流式处理范式的规范定义，介绍了它的一些常见属性，并将它与其他编程范式进行了比较。

然后列举了 3 个基于 Kafka Streams 开发的应用程序，以此来解释一些非常重要的流式处理概念。

在详述了这些示例之后，我们给出了 Kafka Streams 的架构概览，并解释了它的内部原理。最后提供了一些流式处理的使用场景，给出了一些用于比较流式处理框架的建议，并以此结束本书。

附录 A　在其他操作系统上安装 Kafka

Kafka 基本上就是一个 Java 应用程序，所以它可以运行在任何一个可以安装 JRE 的系统上。不过，它针对 Linux 系统进行了优化，因此可以获得最好的性能。如果让它运行在其他系统上，有可能会出现与特定操作系统相关的问题。所以，在桌面操作系统上进行 Kafka 开发或测试时，最好能够让它运行在虚拟机里，这个虚拟机最好能与生产环境的配置相匹配。

A.1　在Windows上安装Kafka

截止到 Windows 10，可以通过两种方式来运行 Kafka。传统的方式是使用本地的 Java 安装包。Windows 10 用户还可以使用 Windows 的 Linux 子系统。推荐使用第二种方式，因为它提供了与生产环境最为相似的配置，所以这里就从这种方式开始说起。

A.1.1　使用Windows的Linux子系统

如果使用的是 Windows 10，就可以通过 Windows 的 Linux 子系统（WSL）来安装原生的 Ubuntu 支持。在本书英文版出版时，微软仍然只是把它当成一个实验特性。它有点像是一个虚拟机，但不像虚拟机那样需要那么多资源，却提供了更丰富的系统集成。

可以按照微软开发者网络 Bash on Ubuntu on Windows（https://msdn.microsoft.com/en-us/commandline/wsl/about）页面所提供的说明来安装 WSL。安装完 WSL 后，需要通过 `apt-get` 来安装 JDK。

```
$ sudo apt-get install openjdk-7-jre-headless
[sudo] password for username:
Reading package lists... Done
Building dependency tree
Reading state information... Done
[...]
done.
$
```

安装完 JDK 后，再根据第 2 章的说明来安装 Kafka。

A.1.2　使用本地Java

如果你使用的是旧版本的 Windows，或者不想使用 WSL 环境，那么可以使用 Windows 的 Java 环境来运行 Kafka。不过要注意的是，这可能会引入 Windows 环境所特有的问题。 Kafka 开发社区可能不会注意到这些问题，因为 Kafka 的开发者很少会在 Windows 上运行 Kafka。

在安装 Zookeeper 和 Kafka 之前，必须有一个 Java 环境。建议安装最新版本的 Oracle Java 8，可以在 Oracle Java SE 下载页面找到最新版的 JDK。下载完整版的 JDK，这样就可以拥有所有的 Java 工具，然后按照指示一步一步安装 JDK 即可。

　注意安装路径

> 在安装 Java 和 Kafka 时，建议把它们安装在不包含空格的目录里。 Windows 允许路径里包含空格，但基于 UNIX 环境设计的应用程序不支持包含空格的路径，而且要指定不同的路径也很困难。在安装 Java 时要谨记这一点。例如，如果安装 JDK 1.8 update 121，可以把它安装在 C:\Java\ jdk1.8.0_121 路径中。

在安装完 Java 后，要设置环境变量。环境变量可以在 Windows 的控制面板里设置，根据操作系统版本的不同，它的位置可能不一样。在 Windows 10 中，先选择“系统和安全”，再选择“系统”，然后单击“计算机名、域和工作组设置”里的“更改设置”按钮，这样就可以打开一个“系统属性”窗口，选择“高级”选项卡，然后单击“环境变量”按钮。这里添加一个名为 JAVA_HOME 的用户变量（图 A-1），并把它的值设为 Java 的安装目录，然后修改 Path 系统变量，往里面添加一个新条目 **%JAVA_HOME%\bin**。保存配置，退出控制面板。

接下来可以安装 Kafka 了。安装包里已经包含了 Zookeeper，所以不需要再单独安装 Zookeeper。当前版本的 Kafka 可以从 http://kafka.apache.org/downloads.html 上下载。在本书英文版出版时，Kafka 的版本是 0.10.1.0，相应的 Scala 版本是 2.11.0。下载的文件经过 GZip 压缩，并通过 tar 来打包，所以需要 Windows 的加压缩工具（如 8Zip）来解压。与在 Linux 系统上安装 Kafka 类似，必须为 Kafka 选择一个解压目录。这里假设 Kafka 被解压到 C:\kafka_2.11-0.10.1.0。

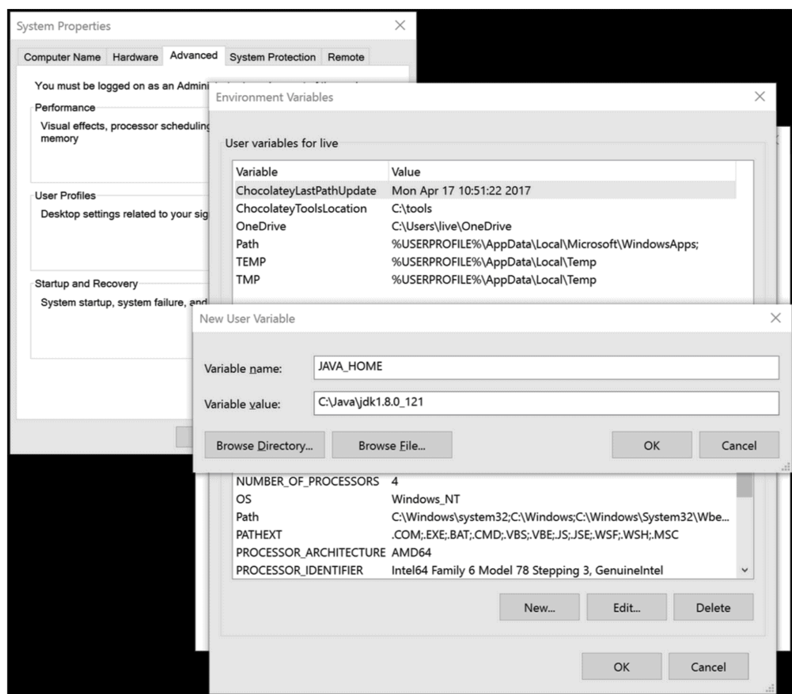


图 **A-1**：添加 **JAVA_HOME** 变量

在 Windows 下运行 Zookeeper 和 Kafka 有一点不一样，因为必须使用 Windows 特有的批处理文件而不是 shell 脚本。批处理文件不支持在后端运行应用程序，所以每一个应用程序都需要一个单独的 shell。先启动 Zookeeper：

```
PS C:\> cd kafka_2.11-0.10.2.0
PS C:\kafka_2.11-0.10.2.0> bin/windows/zookeeper-server-start.bat C:
\kafka_2.11-0.10.2.0\config\zookeeper.properties
[2017-04-26 16:41:51,529] INFO Reading configuration from: C:
\kafka_2.11-0.10.2.0\config\zookeeper.properties (org.apache.zoo
keeper.server.quorum.QuorumPeerConfig)
[...]
[2017-04-26 16:41:51,595] INFO minSessionTimeout set to -1 (org.apache.zoo
keeper.server.ZooKeeperServer)
[2017-04-26 16:41:51,596] INFO maxSessionTimeout set to -1 (org.apache.zoo
keeper.server.ZooKeeperServer)
[2017-04-26 16:41:51,673] INFO binding to port 0.0.0.0/0.0.0.0:2181
(org.apache.zookeeper.server.NIOServerCnxnFactory)
```

在 Zookeeper 开始运行之后，打开另一个窗口来启动 Kafka：

```
PS C:\> cd kafka_2.11-0.10.2.0
PS C:\kafka_2.11-0.10.2.0> .\bin\windows\kafka-server-start.bat C:
\kafka_2.11-0.10.2.0\config\server.properties
[2017-04-26 16:45:19,804] INFO KafkaConfig values:
[...]
[2017-04-26 16:45:20,697] INFO Kafka version : 0.10.2.0 (org.apache.kafka.
mon.utils.AppInfoParser)
[2017-04-26 16:45:20,706] INFO Kafka commitId : 576d93a8dc0cf421
(org.apache.kafka.common.utils.AppInfoParser)
[2017-04-26 16:45:20,717] INFO [Kafka Server 0], started (kafka.server.Kaf
Server)
```

A.2 在MacOS上安装Kafka

苹果的 MacOS 运行在 Darwin 上，一个基于 FreeBSD 的 UNIX 操作系统。也就是说，在 MacOS 上运行应用程序与在 UNIX 系统上是差不多的，所以在 MacOS 上安装为 UNIX 而设计的应用程序（如 Kafka）不会有太大难度。可以通过包管理器（如 Homebrew）来安装 Java 和 Kafka，也可以手动安装它们，这样可以自由选择版本。

A.2.1 使用Homebrew

如果已经在 MacOS 上安装了 Homebrew（https://brew.sh），就可以用它来安装 Kafka。这样可以确保先安装 Java，然后再安装 Kafka 0.10.2.0（在本书英文版出版时）。

如果还没有安装 Homebrew，请先按照 Homebrew 安装页面（https://docs.brew.sh/Installation.html）上提供的步骤安装 Homebrew，然后用它来安装 Kafka。Homebrew 会确保先安装所有的依赖项，包括 Java。

```
$ brew install kafka
==> Installing kafka dependency: zookeeper
[...]
==> Summary
/usr/local/Cellar/kafka/0.10.2.0: 132 files, 37.2MB
$
```

Homebrew 会将 Kafka 安装到 /usr/local/Cellar 目录，不过文件会被链接到其他目录。

- 二进制文件和脚本文件在 /usr/local/bin 目录下。

- Kafka 配置文件在 /usr/local/etc/kafka 目录下。
- Zookeeper 配置文件在 /usr/local/etc/zookeeper 目录下。
- log.dirs（Kafka 的数据目录）被设置为 /usr/local/var/lib/kafka-logs。

安装完毕后，就可以启动 Zookeeper 和 Kafka（把 Kafka 运行在前台）了：

```
$ /usr/local/bin/zkServer start
JMX enabled by default
Using config: /usr/local/etc/zookeeper/zoo.cfg
Starting zookeeper ... STARTED
$ kafka-server-start.sh /usr/local/etc/kafka/server.properties
[...]
[2017-02-09 20:48:22,485] INFO [Kafka Server 0], started (kafka.server.Kaf
Server)
```

A.2.2 手动安装

在 MacOS 上手动安装 Kafka 与在 Windows 上类似，需要先安装 JDK。可以从 Oracle Java SE 下载页面上找到 MacOS 的 JDK 版本，然后下载 Kafka。这里假设下载的 Kafka 被解压到 /usr/local/kafka_2.11-0.10.2.0 目录。

在 MacOS 上启动 Zookeeper 和 Kafka 与在 Linux 上类似，只不过要确保设置了正确的 JAVA_HOME 变量：

```
$ export JAVA_HOME=`/usr/libexec/java_home`
$ echo $JAVA_HOME
/Library/Java/JavaVirtualMachines/jdk1.8.0._131.jdk/Contents/Home
$ /usr/local/kafka_2.11-0.10.2.0/bin/zookeeper-server-start.sh -daemon /us
local/kafka_2.11-0.10.2.0/config/zookeeper.properties
$ /usr/local/kafka_2.11-0.10.2.0/bin/kafka-server-start.sh /usr/local/etc/
server.properties
[2017-04-26 16:45:19,804] INFO KafkaConfig values:
[...]
[2017-04-26 16:45:20,697] INFO Kafka version : 0.10.2.0 (org.apache.kafka.
mon.utils.AppInfoParser)
[2017-04-26 16:45:20,706] INFO Kafka commitId : 576d93a8dc0cf421
(org.apache.kafka.common.utils.AppInfoParser)
[2017-04-26 16:45:20,717] INFO [Kafka Server 0], started (kafka.server.Kaf
Server)
```

作者介绍

Neha Narkhede，Confluent（Kafka 背后的公司）的联合创始人兼工程主管。在创立 Confluent 之前，Neha 在 LinkedIn 领导流式基础设施团队，基于 Kafka 和 Apache Samza 构建 PB 级别的流式基础设施。Neha 擅长构建大型可伸缩的分布式系统，是 Kafka 的最初作者之一。她曾经在 Oracle 从事数据库搜索方面的工作，拥有佐治亚理工大学计算机科学硕士学位。

Gwen Shapira，Confluent 产品经理，同时也是 Kafka 项目的 PMC（Apache 项目管理委员会）成员。她实现了 Kafka 与 Apache Flume 的集成，同时也是 Apache Sqoop 的贡献者之一。Gwen 拥有 15 年与客户共同设计可伸缩数据架构的经验。她曾经是 Cloudera 的一名软件工程师、Pythian 的高级顾问、Oracle ACE 总监，以及 NoCOUG 的董事会成员。Gwen 经常在各种行业大会上演讲，为多个行业博客撰写文章，包括 O'Reilly Radar。

Todd Palino，LinkedIn 的高级网站可靠性工程师，负责部署和维护大型的 Kafka、 Zookeeper 和 Samza 平台。他负责架构、日常运维和工具部署，包括创建高级的监控和通知系统。Todd 是开源项目 Burrow（一个 Kafka 消费者监控工具）的开发者，经常在行业大会和技术讲座上分享 Kafka 相关的经验。Todd 有超过 20 年的基础设施服务相关经验，在加入 LinkedIn 之前，他是 Verisign 的一名系统工程师，负责实施服务管理自动化，包括 DNS、网络和硬件的管理，并在公司范围内管理硬件和软件标准。

封面介绍

本书封面上的动物是一只蓝翅笑翠鸟。笑翠鸟属于翠鸟科，生活在新几内亚南部和澳大利亚北部。

雄性笑翠鸟拥有五颜六色的羽毛，翅膀和尾部的羽毛是蓝色的，雌性笑翠鸟的尾部则是红棕色，带有黑色的条纹。雄性和雌性笑翠鸟都有奶油色的腹部，伴有棕色的条纹，它们的虹膜是白色的。成年笑翠鸟的体型要比其他翠鸟小一些，平均身长 38 到 43 厘米，体重在 260 克到 330 克之间。

蓝翅笑翠鸟偏爱肉食，捕食对象随季节稍有不同。在夏季，它们捕捉蜥蜴、昆虫和青蛙为食，而在干燥的季节，它们则多以小龙虾、鱼、啮齿类动物和小型的鸟类为食。不过，在以鸟类为食的食物链里，红鹰和红褐色的猫头鹰喜欢以蓝翅笑翠鸟为食。

9 月到 12 月是蓝翅笑翠鸟的繁殖季节，它们把巢穴筑在树的上方，通过社区协作的方式养育下一代，一对笑翠鸟至少会得到另外一只笑翠鸟的帮助。它们一般会花大概 26 天的时间来孵蛋，每次孵 3 到 4 只蛋。雏鸟如果能够正常破壳而出，那么大概 36 天后就会长出羽毛。早出生的雏鸟会在第一周内尝试杀死更小的雏鸟。成年笑翠鸟会花上 6 到 8 周的时间来训练那些存活下来的小鸟怎样捕食。

O'Reilly 封面上的很多动物都是濒危物种，它们对整个世界都很重要。如果你想为保护动物做些贡献，可以访问 animals.oreilly.com。

封面图片来自 English Cyclopedia。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_interview，讲述码农精彩人生
- 微信 图灵教育：turingbooks

091507240605ToBeReplacedWithUserId

版权信息

书名：Flink基础教程

作者：[美] 埃伦 • 弗里德曼 [希] 科斯塔斯 • 宙马斯

译者：王绍翾

ISBN：978-7-115-49006-3

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

091507240605ToBeReplacedWithUserId

版权声明
O'Reilly Media, Inc. 介绍
业界评论
前言
如何阅读本书
排版约定
电子书
第 1 章　为何选择 Flink
1.1　流处理欠佳的后果
1.1.1　零售业和市场营销
1.1.2　物联网
1.1.3　电信业
1.1.4　银行和金融业
1.2　连续事件处理的目标
1.3　流处理技术的演变
1.4　初探Flink
批处理与流处理
1.5　生产环境中的Flink
1.5.1　布衣格电信
1.5.2　其他案例
1.6　Flink的适用场景
第 2 章　流处理架构
2.1　传统架构与流处理架构
2.2　消息传输层和流处理层
2.3　消息传输层的理想功能
2.3.1　兼具高性能和持久性
2.3.2　将生产者和消费者解耦
2.4　支持微服务架构的流数据
2.4.1　数据流作为中心数据源
2.4.2　欺诈检测：流处理架构用例
2.4.3　给开发人员带来的灵活性
2.5　不限于实时应用程序

2.6 流的跨地域复制
第 3 章 Flink 的用途
3.1 不同类型的正确性
3.1.1 符合产生数据的自然规律
3.1.2 事件时间
3.1.3 发生故障后仍保持准确
3.1.4 及时给出所需结果
3.1.5 使开发和运维更轻松
3.2 分阶段采用Flink
第 4 章 对时间的处理
4.1 采用批处理架构和Lambda架构计数
4.2 采用流处理架构计数
流处理系统中的批处理
4.3 时间概念
4.4 窗口
4.4.1 时间窗口
4.4.2 计数窗口
4.4.3 会话窗口
4.4.4 触发器
4.4.5 窗口的实现
4.5 时空穿梭
4.6 水印
水印是如何生成的
4.7 真实案例：爱立信公司的Kappa架构
第 5 章 有状态的计算
5.1 一致性
5.2 检查点：保证exactly-once
5.3 保存点：状态版本控制
5.4 端到端的一致性和作为数据库的流处理器
5.5 Flink的性能
5.5.1 Yahoo! Streaming Benchmark
5.5.2 变化1：使用Flink状态

5.5.3　变化2：改进数据生成器并增加吞吐量
5.5.4　变化3：消除网络瓶颈
5.5.5　变化4：使用MapR Streams
5.5.6　变化5：增加key基数
5.6　结论
第 6 章　批处理：一种特殊的流处理
6.1　批处理技术
6.2　案例研究：Flink作为批处理器
附录　其他资源
进一步使用Flink
关于时间和窗口的更多内容
关于状态和检查点的更多内容
用Flink进行批处理
Flink用例和用户故事
流处理架构
消息传输：Kafka
消息传输：MapR Streams
Ted Dunning和Ellen Friedman的部分著作
关于作者

版权声明

© 2016 by Ellen Friedman and Kostas Tzoumas.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2018. Authorized translation of the English edition, 2018 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2016。

简体中文版由人民邮电出版社出版，2018。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还

将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，O'Reilly 的每一项产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 **O'Reilly** 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 **Yogi Berra** 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，**Tim** 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

前言

近几年，许多人开始对如何分析大规模系统中的流数据感兴趣，部分原因是，在某些场景下对实时数据进行实时分析显得非常有价值和吸引力。然而，通过低延迟的应用程序及时获得有用的信息，只是高性能流处理带来的众多好处之一。

本书介绍的 Apache Flink（以下简称 Flink）作为一种高度创新的开

源流处理器，具备惊人的潜力，能够帮助你在以流为基础的各种计算中获益。Flink 不仅可以真正实现实时的容错性分析，还可以分析历史数据，并且极大地简化数据处理流程。最让人惊喜的是，Flink 用同一种底层技术来实现流处理和批处理。它拥有完备的语义和强大的性能，这使得应用程序的开发变得简单，其架构也使得应用程序的维护变得容易。

本书将全面介绍 Flink 的功能，并且讲解常见的使用方法，包括如何在生产环境中使用它。Flink 社区由来自世界各地的开发人员和用户组成，整个社区十分活跃，并且成长迅速。第一届 Flink 专属研讨会定名为 Flink Forward，于 2015 年 10 月在德国柏林举行，第二届于 2016 年 9 月举行。还有各种线下聚会在全球范围内举行，新的 Flink 用例在聚会中被大家广泛讨论。

如何阅读本书

本书对技术人员和非技术人员都有帮助。对于本书所讲解的设计理念和功能，你并不需要具备特殊技能或者拥有流处理经验就能理解，但是如果对大数据系统有一定的了解，将会使阅读获得更好的效果。如果需要尝试运行本书中的示例代码，则需要具备 Java 或者 Scala 的经验。本书会清楚地讲解示例背后的核心概念，即使不懂代码也并不影响阅读。

第 1~3 章阐述 Flink 是基于哪些需求被开发出来的，以及它如何满足这些需求；还会介绍流处理架构的优势，以及 Flink 的整体设计。第 4 章至附录对 Flink 的功能进行更深层的技术性阐释。

排版约定



该图标表示一般性注解。



该图标表示提示或建议。

电子书

扫描如下二维码，即可购买本书电子版。



第 1 章　为何选择 Flink

> 人们对某件事的正确理解往往来自基于有效论据的结论。要获得这样的结论，最有效的方法就是沿着事件发生的轨迹进行分析。

许多系统都会产生连续的事件流，如行驶中的汽车发射出 GPS 信号，金融交易，移动通信基站与繁忙的智能手机进行信号交换，网络流量，机器日志，工业传感器和可穿戴设备的测量结果，等等。如果能够高效地分析大规模流数据，我们对上述系统的理解将会更清楚、更快速。简而言之，流数据更真实地反映了我们的生活方式。

因此，我们自然希望将数据用事件流的方式收集起来并加以处理。但直到目前，这并不是整个行业的标准做法。流处理并非全新的概念，但它确实是一项专业性强且极具挑战性的技术。实际上，企业常见的数据架构仍旧假设数据是有头有尾的有限集。这个假设存在的大部分原因在于，与有限集匹配的数据存储及处理系统建起来比较简单。但是，这样做无疑给那些天然的流式场景人为地加了限制。

我们渴望按照流的方式处理数据，但要做好很困难；随着大规模数据在各行各业中出现，难度越来越大。这是一个属于物理学范畴的难

题：在大型分布式系统中，数据一致性和对事件发生顺序的理解必然都是有限的。伴随着方法和技术的演化，我们尽可能使这种局限性不危及商业目标和运营目标。

在这样的背景下，Apache Flink（以下简称 Flink）应运而生。作为在公共社区中诞生的开源软件，Flink 为大容量数据提供流处理，并用同一种技术实现批处理。

在 Flink 的开发过程中，开发人员着眼于避免其他流处理方法不得不在高效性或者易用性方面所做的妥协。

本书将讨论流处理的一些潜在好处，从而帮助你确定以流为基础的数据处理方法是否适合你自己的商业目标。流处理的一些数据来源以及适用场景可能会让你感到意外。此外，本书还将帮助你理解 Flink 的技术以及这些技术如何克服流处理面临的困难。

本章将介绍人们希望通过分析流数据获得什么，以及在大规模流数据分析过程中面临的困难。本章是关于 Flink 的入门介绍，你可以看到人们平常（包括在生产环境中）是怎么使用它的。

1.1　流处理欠佳的后果

谁需要和流数据打交道呢？首先映入脑海的是从事传感器测量和金融交易的工作人员。对于他们来说，流处理非常有用。但是流数据来源非常广泛，两个常见的例子是：网站获得的能够反映用户行为的点击流数据，以及私有数据中心的机器日志。事实上，流数据来源无处不在，但是从连续事件中获得数据并不意味着可以在批量计算中使用这些数据。如今，处理大规模流数据的新技术正在改变这一状况。

如果说处理大规模流数据是一个历史性难题，我们为什么还要不厌其烦地尝试打造更好的流处理系统呢？在介绍支持流处理的新架构及新技术之前，我们先来谈谈不能很好地处理流数据会有什么后果。

1.1.1　零售业和市场营销

在现代零售业中，网站点击量就代表了销量。网站获得的点击数据可能是大量、连续、不均匀的。用以往的技术很难处理好如此规模的数据。仅是构建批量系统处理这些数据流 1 就很有挑战性：结果很可能是需要一个庞大且复杂的系统。并且，传统的做法还会带来数据丢失、延迟、错误的聚合结果等问题。这样的结果怎能对商业领域有所

帮助呢？

> 1在本书中，“数据流”是指由连续数据组成的流；“流数据”是指数据流中的数据。——译者注

假设你正在向首席执行官汇报上一季度的销售数据，你肯定不想事后因为使用了不准确的数据而不得不向首席执行官更正汇报结果。如果不能良好地处理点击数据，你很可能对网站点击量进行不准确的计算，这将导致广告投放报价和业绩数字不准确。

航空旅客服务业面临同样的挑战：航空公司需要快速、准确地处理从各种渠道获得的大量数据。例如，当为一名旅客办理登机手续时，需要对该旅客的机票预订数据进行核对，还需要核对行李处理信息、航班状态信息和账单信息。如果没有强大的技术来支持流处理，这种规模的数据是很难不出错的。近几年，美国四大航空公司中有三家都出现了大面积的服务中断，这几次故障都可以归咎于大规模实时数据处理失败。

当然，很多相关问题（如怎样避免重复预订酒店或演唱会门票），一般都能够通过有效的数据库操作来解决，但是这种操作相当费钱，也费精力。尤其当数据量增加时，成本会飙升，并且在某些情况下，数据库的反应速度会变得特别慢。由于缺乏灵活性，开发速度受到影响，项目在庞大又复杂或者不断发生变化的系统中进展缓慢。想要在大型系统中处理流数据，并且在保持一致性的同时有效地控制成本，难度非常大。

幸运的是，现代的流处理器经常可以用新的方式解决这些问题，这使得实时处理大规模数据的成本更低。流处理还激发了新的尝试，比如构建一个系统，该系统能够基于顾客当下购买的商品实时给出相关的建议，看看他们是否还需要买一些别的商品。这不代表流处理器替代了数据库（远远不能替代），而是说在数据库处理不好时，流处理器提供了更好的解决方案。这样做也使数据库得以解脱，不用再参与对当前业务状态的实时分析。第 2 章在介绍流处理架构时将对这一转变做更深入的讲解。

1.1.2　物联网

物联网是流数据被普遍应用的领域。在物联网中，低延迟的数据传输和处理，以及准确的数据分析通常很关键。各类仪器中的传感器频繁地获得测量数据，并将它们以流的形式传输至数据中心。在数据中心内，实时或者接近实时的应用程序将更新显示板，运行机器学习模

型，发布警告，并就许多不同的服务项目提供反馈。

交通运输业也体现了流处理的重要性。举例来说，先进的列车系统依靠的是传感器测量数据，这些数据从轨道传至列车，再从列车传至沿途的传感器；与此同时，报告也被发送回控制中心。测量数据包括列车的速度和位置，以及轨道周边的状况。如果流数据没有被正确处理，调整意见和警告就不能相应产生，从而也就不能通过对危险状况做出反应来避免事故发生。

另一个例子是“智能”汽车，或称联网汽车，它们通过移动网络将数据传输回制造商。在有些国家（北欧国家、法国和英国，美国则刚开始），联网汽车甚至可以将信息传给保险公司；如果是赛车，信息还可以通过射频链路传送至维修站进行分析。此外，一些智能手机应用程序还支持数百万司机共享实时路况信息。



图 **1-1**：许多情况都需要考虑数据的时效性，包括使用物联网数据的交通运输业。供数百万司机共享的实时路况信息依靠的是对流数据及时地进行合理和准确的分析（图片来源：**©2016** 弗里德曼）

物联网对公用事业也有影响。相关公司已经开始安装智能计量表，以替换每个月需要人工读数的旧表。智能计量表可以定期将用电量反馈给公司（例如每 15 分钟一次）。有些公司正在尝试每 30 秒就进行一次测量。使用智能计量表的这一转变带来了大量的流数据，同时也获得了大量的潜在收益。其中一个好处就是通过机器学习模型来检测设备故障或者窃电等使用异常。如果不能对流数据进行高吞吐、低延迟和准确的处理，这些新的目标都无法实现。

如果流处理做得不好，其他物联网项目也会遭殃。大型设备，比如风力涡轮机、生产设备和钻井泵，都依赖对传感器测量数据的分析来获得故障警告。如果不能及时地处理好这些设备的流数据，将可能付出高昂的代价，甚至导致灾难性后果。

1.1.3 电信业

电信业是一个特殊的例子，它广泛地应用了基于各种目的而产生的跨地域的事件流数据。如果电信公司不能很好地处理流数据，就不能在某个移动通信基站出现流量高峰前预先将流量分配给其他的基站，也不能在断电时快速做出反应。通过处理流数据来进行异常检测，如检测通话中断或者设备故障，对于电信业来说至关重要。

1.1.4 银行和金融业

因为流处理做得不好而给银行以及金融业带来的潜在问题是极其显著的。从事零售业务的银行不希望客户交易被延迟或者因为错误统计而造成账户余额出错。曾有一个说法叫作“银行家工作时间”，指的就是银行需要在下午早早关门进行结算，这样才能保证第二天营业之前算出准确的账。这种批量作业的营业模式早已消失。如今，交易和报表都必须快速且准确地生成；有些新兴的银行甚至提供实时的推送通知，以及随时随地访问手机银行的服务。在全球化经济中，能够提供 24 小时服务变得越来越重要。

那么，如果缺少能够灵敏地实时检测出用户行为异常的应用程序，会对金融机构带来什么后果呢？信用卡欺诈检测需要及时的监控和反馈。对异常登录的检测能发现钓鱼式攻击，从而避免巨大的损失。

> 在许多情况下，人们希望用低延迟或者实时的流处理来获得数据的高时效性，前提是流处理本身是准确且高效的。

1.2 连续事件处理的目标

能够以非常低的延迟处理数据，这并不是流处理的唯一优势。人们希望流处理不仅做到低延迟和高吞吐，还可以处理中断。优秀的流处理技术应该能使系统在崩溃之后重新启动，并且产出准确的结果；换句话说，优秀的流处理技术可以容错，而且能保证 exactly-once[2]。

[2]对 exactly-once 的解释，详见 5.1 节。——编者注

与此同时，获得这种程度的容错性所采用的技术还需要在没有数据错误的情况下不产生太大的开销。这种技术需要能够基于事件发生的时间（而不是随意地设置处理间隔）来保证按照正确的顺序跟踪事件。对于开发人员而言，不论是写代码还是修正错误，系统都要容易操作和维护。同样重要的是，系统生成的结果需要与事件实际发生的顺序一致，比如能够处理乱序事件流（一个很不幸但无法避免的事实），以及能够准确地替换流数据（在审计或者调试时很有用）。

1.3 流处理技术的演变

分开处理连续的实时数据和有限批次的数据，可以使系统构建工作变得更加简单，但是这种做法将管理两套系统的复杂性留给了系统用户：应用程序的开发团队和 DevOps 团队需要自己使用并管理这两套系统。

为了处理这种情况，有些用户开发出了自己的流处理系统。在开源世界里，Apache Storm 项目（以下简称 Storm）是流处理先锋。Storm 最早由 Nathan Marz 和创业公司 BackType（后来被 Twitter 收购）的一个团队开发，后来才被 Apache 软件基金会接纳。Storm 提供了低延迟的流处理，但是它为实时性付出了一些代价：很难实现高吞吐，并且其正确性没能达到通常所需的水平。换句话说，它并不能保证 exactly-once；即便是它能够保证的正确性级别，其开销也相当大。

Lambda 架构概述：优势和局限性

对低成本规模化的需求促使人们开始使用分布式文件系统，例如 HDFS 和基于批量数据的计算系统（MapReduce 作业）。但是这种系统很难做到低延迟。用 Storm 开发的实时流处理技术可以帮助解决延迟性的问题，但并不完美。其中的一个原因是，Storm 不支持 exactly-once 语义，因此不能保证状态数据的正确性，另外它也不支持基于事件时间的处理。有以上需求的用户不得不在自己的应用程序代码中加入这些功能。

后来出现了一种混合分析的方法，它将上述两个方案结合起来，既保证低延迟，又保障正确性。这个方法被称作 Lambda 架构，它通过批量 MapReduce 作业提供了虽有些延迟但是结果准确的计算，同时通过

Storm 将最新数据的计算结果初步展示出来。

Lambda 架构是构建大数据应用程序的一种很有效的框架，但它还不够好。举例来说，基于 MapReduce 和 HDFS 的 Lambda 系统有一个长达数小时的时间窗口，在这个窗口内，由于实时任务失败而产生的不准确的结果会一直存在。Lambda 架构需要在两个不同的 API（application programming interface，应用程序编程接口）中对同样的业务逻辑进行两次编程：一次为批量计算的系统，一次为流式计算的系统。针对同一个业务问题产生了两个代码库，各有不同的漏洞。这种系统实际上非常难维护。

若要依靠多个流事件来计算结果，必须将数据从一个事件保留到下一个事件。这些保存下来的数据叫作计算的状态。准确处理状态对于计算结果的一致性至关重要。在故障或中断之后能够继续准确地更新状态是容错的关键。

在低延迟和高吞吐的流处理系统中维持良好的容错性是非常困难的，但是为了得到有保障的准确状态，人们想出了一种替代方法：将连续事件中的流数据分割成一系列微小的批量作业。如果分割得足够小（即所谓的微批处理作业），计算就几乎可以实现真正的流处理。因为存在延迟，所以不可能做到完全实时，但是每个简单的应用程序都可以实现仅有几秒甚至几亚秒的延迟。这就是在 Spark 批处理引擎上运行的 Apache Spark Streaming（以下简称 Spark Streaming）所使用的方法。

更重要的是，使用微批处理方法，可以实现 exactly-once 语义，从而保障状态的一致性。如果一个微批处理作业失败了，它可以重新运行。这比连续的流处理方法更容易。Storm Trident 是对 Storm 的延伸，它的底层流处理引擎就是基于微批处理方法来进行计算的，从而实现了 exactly-once 语义，但是在延迟性方面付出了很大的代价。

然而，通过间歇性的批处理作业来模拟流处理，会导致开发和运维相互交错。完成间歇性的批处理作业所需的时间和数据到达的时间紧密耦合，任何延迟都可能导致不一致（或者说错误）的结果。这种技术的潜在问题是，时间由系统中生成小批量作业的那一部分全权控制。Spark Streaming 等一些流处理框架在一定程度上弱化了这一弊端，

但还是不能完全避免。另外，使用这种方法的计算有着糟糕的用户体验，尤其是那些对延迟比较敏感的作业，而且人们需要在写业务代码时花费大量精力来提升性能。

为了实现理想的功能，人们继续改进已有的处理器（比如 Storm Trident 的开发初衷就是试图克服 Storm 的局限性）。当已有的处理器不能满足需求时，产生的各种后果则必须由应用程序开发人员面对和解决。以微批处理方法为例，人们往往期望根据实际情况分割事件数据，而处理器只能根据批量作业时间（恢复间隔）的倍数进行分割。当灵活性和表现力都缺乏的时候，开发速度变慢，运维成本变高。

于是，Flink 出现了。这一数据处理器可以避免上述弊端，并且拥有所需的诸多功能，还能按照连续事件高效地处理数据。Flink 的一些功能如图 1-2 所示。

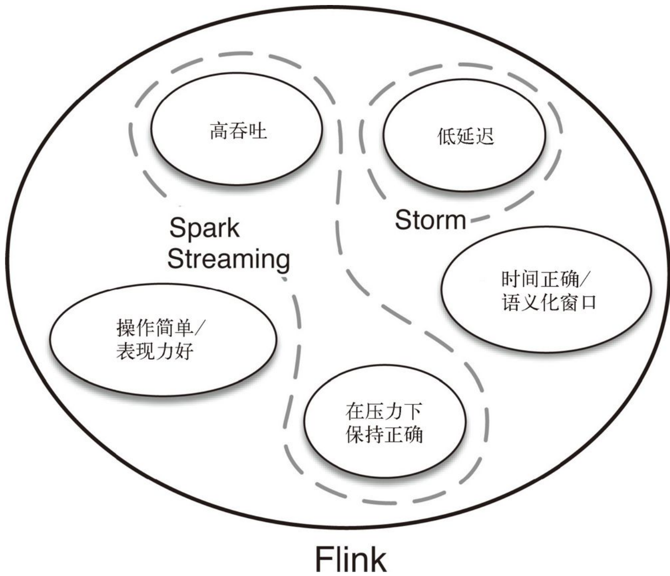


图 **1-2**：**Flink** 的一个优势是，它拥有诸多重要的流式计算功能。其他项目为了实现这些功能，都不得不付出代价。比如，**Storm** 实现了低延迟，但是在作者撰写本书时还做不到高吞吐，也不能在故障发生时准确地处理计算状态；**Spark Streaming** 通过采用微批处理方法实现了高吞吐和容错性，但是牺牲了低延迟和实时处理能力，也不能使窗口与自然时间相匹配，并且表现力欠佳

与 Storm 和 Spark Streaming 类似，其他流处理技术同样可以提供一

些有用的功能，但是没有一个像 Flink 那样功能如此齐全。举例来说，Apache Samza（以下简称 Samza）是早期的一个开源流处理器，它不仅没能实现 exactly-once 语义，而且只能提供底层的 API；同样，Apache Apex 提供了与 Flink 相同的一些功能，但不全面（比如只提供底层的 API，不支持事件时间，也不支持批量计算）。这些项目没有一个能和 Flink 在开源社区的规模上相提并论。

下面来了解 Flink 是什么，以及它是如何诞生的。

1.4　初探Flink

Flink 的主页 3 在其顶部展示了该项目的理念："Apache Flink 是为分布式、高性能、随时可用以及准确的流处理应用程序打造的开源流处理框架。"Flink 不仅能提供同时支持高吞吐和 exactly-once 语义的实时计算，还能提供批量数据处理，这让许多人感到吃惊。鱼与熊掌并非不可兼得，Flink 用同一种技术实现了两种功能。

> 3http://flink.apache.org

这个顶级的 Apache 项目是怎么诞生的呢？Flink 起源于 Stratosphere 项目，Stratosphere 是在 2010~2014 年由 3 所地处柏林的大学和欧洲的一些其他的大学共同进行的研究项目。当时，这个项目已经吸引了一个较大的社区，一部分原因是它出现在了若干公共开发者研讨会上，比如在柏林举办的 Berlin Buzzwords，以及在科隆举办的 NoSQL Matters，等等。强大的社区基础是这个项目适合在 Apache 软件基金会中孵化的一个原因。

2014 年 4 月，Stratosphere 的代码被复制并捐献给了 Apache 软件基金会，参与这个孵化项目的初始成员均是 Stratosphere 系统的核心开发人员。不久之后，创始团队中的许多成员离开大学并创办了一个公司来实现 Flink 的商业化，他们为这个公司取名为 data Artisans。在孵化期间，为了避免与另一个不相关的项目重名，项目的名称也发生了改变。Flink 这个名字被挑选出来，以彰显这种流处理器的独特性：在德语中，flink 一词表示快速和灵巧。项目采用一只松鼠的彩色图案作为 logo，这不仅因为松鼠具有快速和灵巧的特点，还因为柏林的松鼠有一种迷人的红棕色。



图 **1-3**：左侧：柏林的红松鼠拥有可爱的耳朵；右侧：**Flink** 的松鼠 **logo** 拥有可爱的尾巴，尾巴的颜色与 **Apache** 软件基金会的 **logo** 颜色相呼应。这是一只 **Apache** 风格的松鼠！

这个项目很快完成了孵化，并在 2014 年 12 月一跃成为 Apache 软件基金会的顶级项目。作为 Apache 软件基金会的 5 个最大的大数据项目之一，Flink 在全球范围内拥有 200 多位开发人员，以及若干公司中的诸多上线场景，有些甚至是世界 500 强的公司。在作者撰写本书的时候，共有 34 个 Flink 线下聚会在世界各地举办，社区大约有 12 000 名成员，还有众多 Flink 演讲者参与到各种大数据研讨会中。2015 年 10 月，第一届 Flink Forward 研讨会在柏林举行。

批处理与流处理

Flink 是如何同时实现批处理与流处理的呢？答案是，Flink 将批处理（即处理有限的静态数据）视作一种特殊的流处理。

Flink 的核心计算构造是图 1-4 中的 Flink Runtime 执行引擎，它是一个分布式系统，能够接受数据流程序并在一台或多台机器上以容错方式执行。Flink Runtime 执行引擎可以作为 YARN（Yet Another Resource Negotiator）的应用程序在集群上运行，也可以在 Mesos 集群上运行，还可以在单机上运行（这对于调试 Flink 应用程序来说非常有用）。

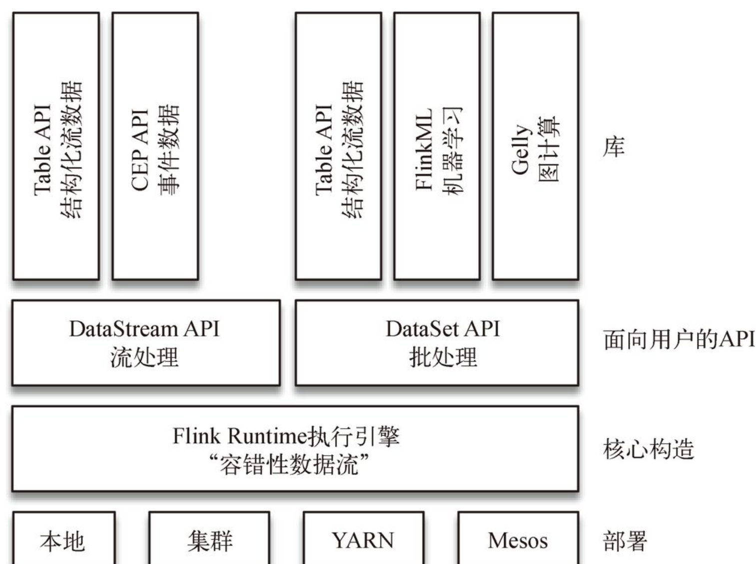


图 **1-4**：**Flink** 技术栈的核心组成部分。值得一提的是，**Flink** 分别提供了面向流处理的接口（**DataStream API**）和面向批处理的接口（**DataSet API**）。因此，**Flink** 既可以完成流处理，也可以完成批处理。**Flink** 支持的拓展库涉及机器学习（**FlinkML**）、复杂事件处理（**CEP**），以及图计算（**Gelly**），还有分别针对流处理和批处理的 **Table API**

能被 Flink Runtime 执行引擎接受的程序很强大，但是这样的程序有着冗长的代码，编写起来也很费力。基于这个原因，Flink 提供了封装在 Runtime 执行引擎之上的 API，以帮助用户更方便地生成流式计算程序。Flink 提供了用于流处理的 DataStream API 和用于批处理的 DataSet API。值得注意的是，尽管 Flink Runtime 执行引擎是基于流处理的，但是 DataSet API 先于 DataStream API 被开发出来，这是因为工业界对无限流处理的需求在 Flink 诞生之初并不大。

DataStream API 可以流畅地分析无限数据流，并且可以用 Java 或者 Scala 来实现。开发人员需要基于一个叫 DataStream 的数据结构来开发，这个数据结构用于表示永不停止的分布式数据流。

Flink 的分布式特点体现在它能够在成百上千台机器上运行，它将大型的计算任务分成许多小的部分，每个机器执行一个部分。Flink 能够自动地确保在发生机器故障或者其他错误时计算能持续进行，或者在修复 bug 或进行版本升级后有计划地再执行一次。这种能力使得开

发人员不需要担心失败。Flink 本质上使用容错性数据流，这使得开发人员可以分析持续生成且永远不结束的数据（即流处理）。

> Flink 解决了许多问题，比如保证了 exactly-once 语义和基于事件时间的数据窗口。开发人员不再需要在应用层解决相关问题，这大大地降低了出现 bug 的概率。

因为不用再在编写应用程序代码时考虑如何解决问题，所以工程师的时间得以充分利用，整个团队也因此受益。好处并不局限于缩短开发时间，随着灵活性的增加，团队整体的开发质量得到了提高，运维工作也变得更容易、更高效。Flink 让应用程序在生产环境中获得良好的性能。尽管相对较新，但是 Flink 已经在生产环境中得到了应用，下一节将做更详细的介绍。

1.5　生产环境中的Flink

本章旨在探讨为何选择 Flink。一个好的方法是听听在生产环境中使用 Flink 的开发人员解释他们选择 Flink 的原因，以及如何使用它。

1.5.1　布衣格电信

布衣格电信（Bouygues Telecom）是法国第三大移动通信运营商，隶属世界 500 强企业布衣格集团。布衣格电信使用 Flink 来进行实时事件处理，每天不间断地分析数十亿条消息。2015 年 6 月，在为 data Artisans 的博客撰写的文章中，Mohamed Amine Abdessemed[4] 描述了布衣格电信的目标以及 Flink 为什么能实现这些目标。

> 4他在布衣格电信负责技术系统。——编者注

> ……布衣格电信最终选择了 Flink，因为它支持真正的流处理——通过上层的 API 和下层的执行引擎都能实时进行流处理，这满足了我们对可编程性和低延迟的需求。此外，使用 Flink，我们的系统得以快速上线，这是其他任何一种方案都做不到的。如此一来，我们就有了更多的人手开发新的业务逻辑。

Mohamed Amine Abdessemed 在于 2015 年 10 月举行的 Flink Forward 研讨会上也做了报告。布衣格电信试图给其工程师实时提供关于用户体验的反馈，让他们了解公司遍布全球的网络正在发生什

么，并从网络的演进和运行的角度掌握发展动向。

为了实现这个目标，他们的团队搭建了一个用来分析网络设备日志的系统，它定义了衡量用户体验质量的各项指标。该系统每天处理 20 亿次事件，要求端到端延迟不超过 200 毫秒（包括由传输层负责的消息发布和由 Flink 操作的数据处理）。这些都在一个仅有 10 个节点（每个节点 1GB 内存）的小集群上完成。布衣格电信还希望这些经过部分处理的数据能被复用，从而在互不干扰的前提下满足各种商业智能分析需求。

该公司打算利用 Flink 的流处理能力来转换和挖掘数据。加工后的数据被推送回消息传输系统，以保证数据可以被不同的用户使用。

相比于其他处理方案，比如在数据进入消息队列之前进行处理，或者将数据分派给消费同一个消息队列的多个应用程序来分头处理，Flink 的处理方案更合适。

布衣格电信利用 Flink 的流处理能力完成了数据处理和数据迁移，它既满足了低延迟的要求，又具有很高的可靠性、可用性，以及易用性。Flink 为调试工作提供了极大的便利，甚至支持切换到本地进行调试。它也支持程序可视化，有利于理解程序的运行原理。此外，Flink 的 API 吸引了很多开发人员和数据科学家。Mohamed Amine Abdessemed 在其文章中还提及布衣格电信的其他团队使用 Flink 解决了不同的问题。

1.5.2　其他案例

King公司

King 公司的游戏非常流行，全世界几乎每时每刻都有人在玩它的在线游戏。作为在线娱乐行业的佼佼者，该公司称自己已经开发了 200 多款游戏，市场覆盖 200 多个国家和地区。

King 公司的工程师曾在一篇博客文章中写道：“我们每月有超过 3 亿的独立用户，每天从不同的游戏和系统中收到 300 亿次事件，基于这么大的数据量做任何流分析都是真正的技术挑战。因此，为我们的数据分析师开发工具来处理如此大规模的流数据，同时保证数据在应用中具有最大的灵活性，这些对于公司而言至关重要。”

King 公司用 Flink 构建的系统让其数据分析师得以实时地获取大量的流数据。Flink 的成熟度给他们留下了深刻的印象。即使面对像 King

公司这样复杂的应用环境，Flink 也能很好地提供支持。

Zalando公司

作为欧洲领先的在线时尚平台，Zalando 公司在全球拥有超过 1600 万的客户。该公司的网站将其组织结构描述为“多个敏捷、自主的小型团队”（换句话说，该公司采用了微服务架构）。

流处理架构为微服务提供了良好的支持。因此，Flink 提供的流处理能力满足了这种工作模式的需求，特别是支持业务流程监控和持续的 ETL5 过程。

> 5ETL 是 Extract、Transform 和 Load 的缩写，即抽取、转换和加载。——编者注

Otto集团

Otto 集团是全球第二大 B2C（business to consumer，企业对顾客电子商务）在线零售商，也是欧洲时尚和生活领域最大的 B2C 在线零售商。

它的商业智能部门在最初开始评估开源流处理平台时，没有找到一种能够符合其要求的平台，所以后来决定开发自己的流处理引擎。但是当试过 Flink 之后，该部门发现 Flink 满足了他们对流处理的所有需求，包括对众包用户代理的鉴定，以及对检索事件的辨识。

ResearchGate

从活跃用户的数量上看，ResearchGate 是最大的学术社交网络。它从 2014 年开始使用 Flink 作为其数据基础设施的一个主要工具，负责批处理和流处理。

阿里巴巴集团

阿里巴巴这个庞大的电子商务集团为买方和卖方提供平台。其在线推荐功能是通过基于 Flink 的系统 Blink 实现的。用户当天所购买的商品可以被用作在线推荐的依据，这是使用像 Flink 这样真正意义上的流处理引擎能够带来的好处之一。并且，这在那些用户活跃度异常高的特殊日期（节假日）尤其重要，也是高效的流处理相较于批处理的优势之一。

1.6 Flink的适用场景

本章开头提出了“为何选择 Flink”这一问题。比这个问题更大的则是“为何要用流数据？”本章解释了一些原因，比如在许多情况下，我们都需要观察和分析连续事件产生的数据。与其说流数据是特别的，倒不如说它是自然的——只不过从前我们没有流处理能力，只能做一些特殊的处理才能真正地使用流数据，比如将流数据攒成批量数据再处理，不然无法进行大规模的计算。使用流数据并不新鲜，新鲜的是我们有了新技术，从而可以大规模、灵活、自然和低成本地使用它们。

Flink 并不是唯一的流处理工具。人们正在开发和改进多种新兴的技术，以满足流处理需求。显然，任何一个团队选择某一种技术都是出于多方面的考虑，包括团队成员的已有技能。但是 Flink 的若干优点、易用性，以及使用它所带来的各种好处，使它变得非常有吸引力。另外，不断壮大且非常活跃的 Flink 社区也暗示着它值得一试。你会发现“为何选择 Flink”这个问题变成了“为何不选择 Flink 呢？”

在深入探讨 Flink 的工作原理之前，我们先来通过第 2 章了解如何设计数据架构才能从流处理中充分获益，以及流处理架构是如何带来诸多好处的。

第 2 章 流处理架构

数据架构设计领域正在发生一场变革，其影响不仅限于实时或近实时的项目。这场变革将基于流的数据处理流程视为整个架构设计的核心，而不是只作为某些专业化工作的基础。了解为何向流处理架构转变，可以帮助我们理解 Flink 和它在现代数据处理中所扮演的角色。

作为新型系统，Flink 扩展了“流处理”这个概念的范围。有了它，流处理不仅指实时、低延迟的数据分析，还指各类数据应用程序。其中，有些应用程序基于流处理器实现，有些基于批处理器实现，有些甚至基于事务型数据库实现。

事实证明，让 Flink 能有效工作的数据架构，恰恰是充分利用流数据的基础。为了帮助你理解，本书将详细介绍如何构建支持 Flink 流处理的管道。在这之前，我们先来看看与传统架构相比，流处理架构有

何优势。

2.1　传统架构与流处理架构

对于后端数据而言，典型的传统架构是采用一个中心化的数据库系统，该系统用于存储事务性数据。换句话说，数据库（SQL 或者 NoSQL）拥有“新鲜”（或者说“准确”）的数据，这些数据反映了当前的业务状态，如系统当前有多少已登录的用户，网站当前有多少活跃用户，以及当前每个用户的账户余额是多少。需要新鲜数据的应用程序都依靠数据库实现。分布式文件系统则用来存储不需要经常更新的数据，它们也往往是大规模批量计算所依赖的数据存储方式。

这种传统架构成功地服务了几十年，但随着大型分布式系统中的计算复杂度不断上升，这种架构已经不堪重负。许多公司经常遇到以下问题。

- 在许多项目中，从数据到达到数据分析所需的工作流程太复杂、太缓慢。
- 传统的数据架构太单一：数据库是唯一正确的数据源，每一个应用程序都需要通过访问数据库来获得所需的数据。
- 采用这种架构的系统拥有非常复杂的异常问题处理方法。当出现异常问题时，很难保证系统还能很好地运行。

传统架构的另一个问题是，需要通过在大型分布式系统中不断地更新来维持一致的全局状态。随着系统规模扩大，维持实际数据与状态数据间的一致性变得越来越困难；流处理架构则少了对这方面的要求，只需要维持本地的数据一致性即可。

作为一种新的选择，流处理架构解决了企业在大规模系统中遇到的诸多问题。以流为基础的架构设计让数据记录持续地从数据源流向应用程序，并在各个应用程序间持续流动。没有一个数据库来集中存储全局状态数据，取而代之的是共享且永不停止的流数据，它是唯一正确的数据源，记录了业务数据的历史。在流处理架构中，每个应用程序都有自己的数据，这些数据采用本地数据库或分布式文件进行存储。

2.2　消息传输层和流处理层

如何有效地实现流处理架构并从 Flink 中获益呢？一个常见的做法是设置消息传输层和流处理层，如图 2-1 所示。

(1) 消息传输层从各种数据源（生产者）采集连续事件产生的数据，并传输给订阅了这些数据的应用程序和服务（消费者）。

(2) 流处理层有 3 个用途：①持续地将数据在应用程序和系统间移动；②聚合并处理事件；③在本地维持应用程序的状态。

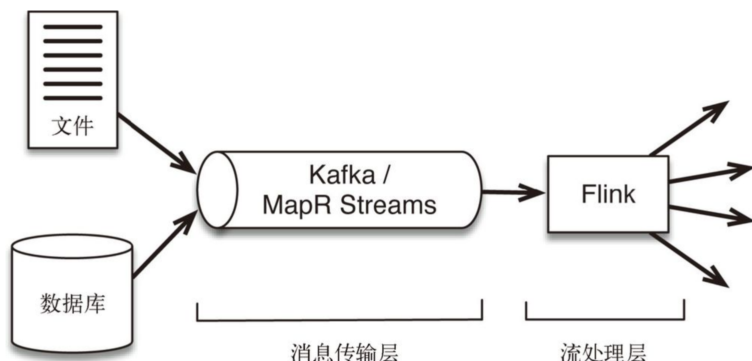


图 **2-1**：**Flink** 项目的架构有两个主要组成部分：消息传输层和由 **Flink** 提供的流处理层。消息传输层负责传输连续事件产生的消息，能够提供消息传输的系统包括 **Kafka** 和 **MapR Streams**。**MapR Streams** 是 **MapR** 融合数据平台的一个主要组成部分，它兼容 **Kafka API**

围绕着实时应用程序产生的兴奋感会将人们的注意力集中到流处理层上，并促使人们思考如何根据具体的项目需求选择流处理技术。除了 Flink 之外，还有其他的流处理工具可供选择（比如 Spark Streaming、Storm、Samza 和 Apex）。本书中的示例都以 Flink 作为流处理器。

事实上，在设计高效的流处理架构时，不仅流处理器的选择会造成架构的巨大差异，消息传输层也很关键。现代系统之所以更容易处理大规模的流数据，其中很大一部分原因就是消息传输方式的改进，以及流处理器与消息传输系统的交互方式的改变。

消息传输层需要具备一些特定的功能。目前来看，有两种技术可以很好地提供所需的功能，它们便是 Kafka 和 MapR Streams。MapR Streams 是 MapR 融合数据平台的一部分，它支持 Kafka API。本书中的示例都假设消息传输层采用 Kafka 或 MapR Streams。

2.3　消息传输层的理想功能

流处理架构的消息传输层需要具备哪些功能呢？

2.3.1　兼具高性能和持久性

消息传输层的一个作用是作为流处理层上游的安全队列——它相当于缓冲区，可以将事件数据作为短期数据保留起来，以防数据处理过程发生中断。直到最近几年，高性能和持久性不可兼得的困境才被打破。人们习惯上认为流数据从消息传输层到流处理层之后就被丢弃：用了就没了。

为了设计新一代的流处理架构，高性能和持久性不可兼得是首先要改变的一个观念。兼具高性能和持久性对于消息传输系统来说至关重要；Kafka 和 MapR Streams 都可以满足这个需求。

具有持久性的好处之一是消息可以重播。这个功能使得像 Flink 这样的处理器能对事件流中的某一部分进行重播和再计算（第 5 章会详细介绍）。正是由于消息传输层和流处理层相互作用，才使得像 Flink 这样的系统有了准确处理和“时空穿梭”（指重新处理数据的能力）的保障，认识到这一点至关重要。

2.3.2　将生产者和消费者解耦

采用高效的消息传输技术，可以从多个源（生产者）收集数据，并使这些数据可供多个服务或应用程序（消费者）使用，如图 2-2 所示。Kafka 和 MapR Streams 把从生产者获得的数据分配给既定的主题。数据源将数据推送给消息队列，消费者（或消费者群组）则拉取数据。事件数据只能基于给定的偏移量从消息队列中按顺序读出。生产者并不向所有消费者自动广播。这一点听起来微不足道，但是对整个架构的工作方式有着巨大的影响。

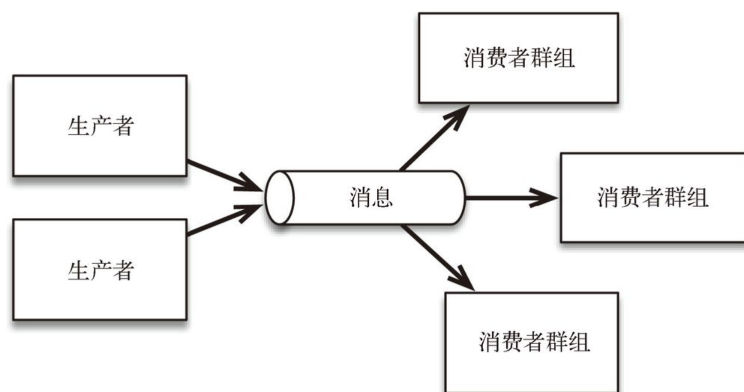


图 **2-2**：在 **Kafka** 和 **MapR Streams** 这样的消息传输工具中，数据的生产者和消费者（**Flink** 应用程序也是其中的一种消费者）是解耦的。到达的消息既可以立刻被使用，也可以稍后被使用。消费者从队列中订阅消息，而不是由生产者向所有消费者广播。在消息到达的时候，消费者不必处于运行状态

这种传输方式——消费者订阅感兴趣的主题——意味着消息立刻到达，但并不需要被立刻处理。在消息到达时，消费者并不需要处于运行状态，而是可以根据自身的需求在任何时间使用数据。这样一来，添加新的消费者和生产者也很容易。采用解耦的消息传输系统很有意义，因为它能支持微服务，也支持将处理步骤中的实现过程隐藏起来，从而允许自由地修改实现过程。

2.4　支持微服务架构的流数据

微服务方法指的是将大型系统的功能分割成通常具有单一目的的简单服务，从而使小型团队可以轻松地构建和维护这些服务。即使是超大型组织，也可以用这种设计实现敏捷。若要使整个系统正常工作，各服务之间因通信而产生的连接必须是轻量级的。

> “（微服务的）目标是给予每个团队一项工作，并授以完成工作的方法，然后就放手。”
>
> 摘自 *Streaming Architecture* 一书的第 3 章，该书由 O'Reilly Media 于 2016 年出版，作者是 Ted Dunning 和 Ellen Friedman。

消息传输系统一方面将生产者和消费者解耦，另一方面又有足够高的

吞吐量，并且能够满足像 Flink 这样的高性能流处理器。这种系统非常适合用于构建微服务。就连接微服务而言，流数据是相对较新的模式，但是它有许多好处，接下来的几小节将进行解释。

2.4.1　数据流作为中心数据源

通过上文的讨论可以看出，流处理架构的核心是使各种应用程序互连在一起的消息队列。流处理器（例如本书介绍的 Flink）从消息队列中订阅数据并加以处理。处理后的数据可以流向另一个消息队列。这样一来，其他应用程序（包括其他 Flink 应用程序）都可以共享流数据。在一些情况下，处理后的数据会被存放在本地数据库中，如图 2-3 所示。

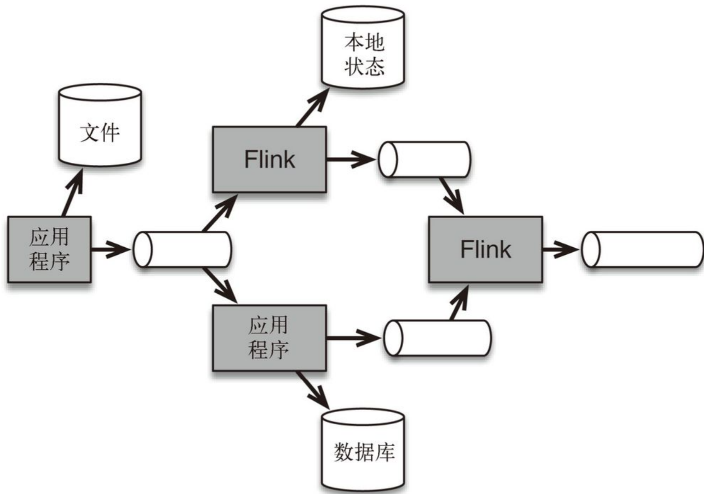


图 **2-3**：在流处理架构中，消息队列（图中以水平圆柱体表示）连接应用程序，并作为新的共享数据源；它们取代了从前的大型集中式数据库。在本例中，**Flink** 被多个应用程序使用。本地化的数据能够根据微服务项目的需要被存储在文件或者数据库中。这种流处理架构的另一个好处是，流处理器（例如 **Flink**）还可以保障数据一致性

> 流处理架构不需要集中式数据库。取而代之的是消息队列，它作为共享数据源，服务于各种不同的消费者。

2.4.2　欺诈检测：流处理架构用例

基于流处理的微服务架构有着强大的灵活性，特别是当同一份数据被用于不同的场景时，其灵活性更为明显。以信用卡服务提供商的欺诈检测项目为例。项目的目标是尽可能快地识别可疑的刷卡行为，从而阻止盗刷，并将损失降到最低。例如，欺诈检测器可以将刷卡速度作为评判依据：在很短的时间内，发生在跨度很大的不同地点，这样的连续交易合理吗？事实上，真正的欺诈检测器会使用几十个（甚至几百个）特征作为评判依据，但是我们仅仅通过分析刷卡速度这一个特征就可以理解许多问题。

图 2-4 展示了流处理架构在欺诈检测项目中的优势。在图中，许多销售终端（POS 机 $1\sim n$）请求欺诈检测器判定是否有欺诈行为。这些来自销售终端的询问需要立即被应答，因此在销售终端与欺诈检测器之间形成了询问与应答的交互。

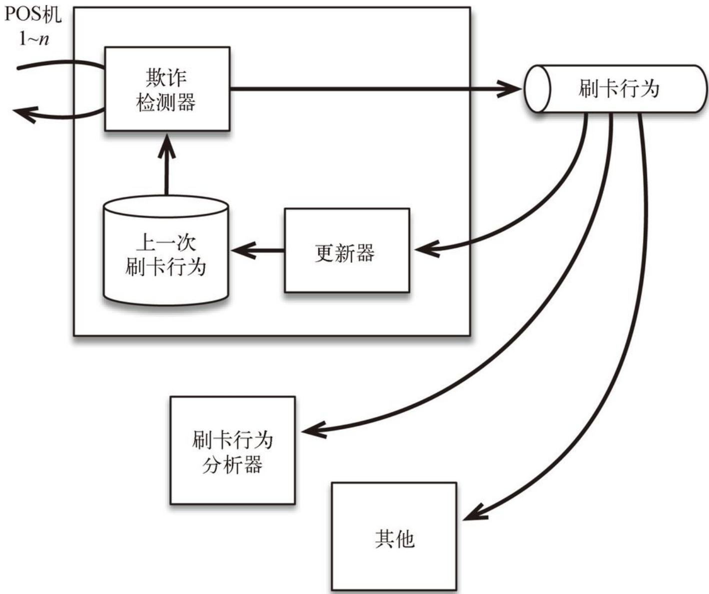


图 **2-4**：欺诈检测可以从基于流处理的微服务架构中受益。在这种架构中，**Flink** 在如下几个部分都非常有用：欺诈检测器、更新器，甚至刷卡行为分析器。值得一提的是，这种架构没有直接在本地数据库中更新刷卡行为的数据，而是将数据放在消息队列里，这些数据还可以不受干扰地被刷卡行为分析器等其他服务使用（图片来源：***Streaming Architecture*** 第 **6** 章）

传统的欺诈检测模型将包含每张信用卡最后一次刷卡地点的文件直接存储在数据库中。但在这样的集中式数据库设计中，其他消费者并不

能轻易使用刷卡行为的数据，因为访问数据库可能会影响欺诈检测系统的正常工作；在没有经过认真仔细的审查之前，其他消费者绝不会被授权更改数据库。这将导致整个流程变慢，因为必须仔细执行各种检查，以避免核心的业务功能受到破坏或影响。

与传统方法相比，图 2-4 所示的流处理架构设计将欺诈检测器的输出发送给外部的消息队列（Kafka 或 MapR Streams），再由如 Flink 这样的流处理器更新数据库，而不是直接将输出发送给数据库。这使得刷卡行为的数据可以通过消息队列被其他服务使用，例如刷卡行为分析器。上一次刷卡行为的数据被存储在本地数据库中，不会被其他服务访问。这样的设计避免了因为增加新的服务而带来的过载风险。

2.4.3　给开发人员带来的灵活性

基于流处理的微服务架构也为欺诈检测系统的开发人员带来了灵活性。假设开发团队正试图改进欺诈检测模型并加以评估。刷卡行为产生的消息流可以被新模型采用，而完全不影响已有的检测器。新增加一个数据消费者的开销几乎可以忽略不计，同时只要合适，数据的历史信息可以保存成任何一种格式，并且使用任意的数据库服务。此外，如果刷卡行为队列中的消息被设计成业务级别的事件，而不是数据库表格的更新，那么消息的形式和内容都会非常稳定。若一定要更改，向前兼容可以避免更改已有的应用程序。

信用卡欺诈检测只是流处理架构的一个用例。流处理架构通过一个合适的消息传输系统（Kafka 或 MapR Streams）和一个多用途、高性能的流处理器（Flink），能支持各种应用程序使用共享数据源，即消息流。

2.5　不限于实时应用程序

虽然低延迟性很重要，但是实时应用程序只是众多流数据消费者中的一种。流数据的应用很广泛，比如，流处理应用程序可以通过订阅消息队列中的流数据来实时更新仪表盘（如图 2-5 中的 A 组消费者）。

持久化的消息可以被重播，这一特性使许多用户获益（如图 2-5 中的 C 组消费者）。在本例中，消息流成为了可审计的日志，或者长期的事件历史。能够重现的历史非常有用，比如可以在工业安全分析中作为预知维护模型的一部分输入，也可以在医学或环境科学领域用于回顾性研究。

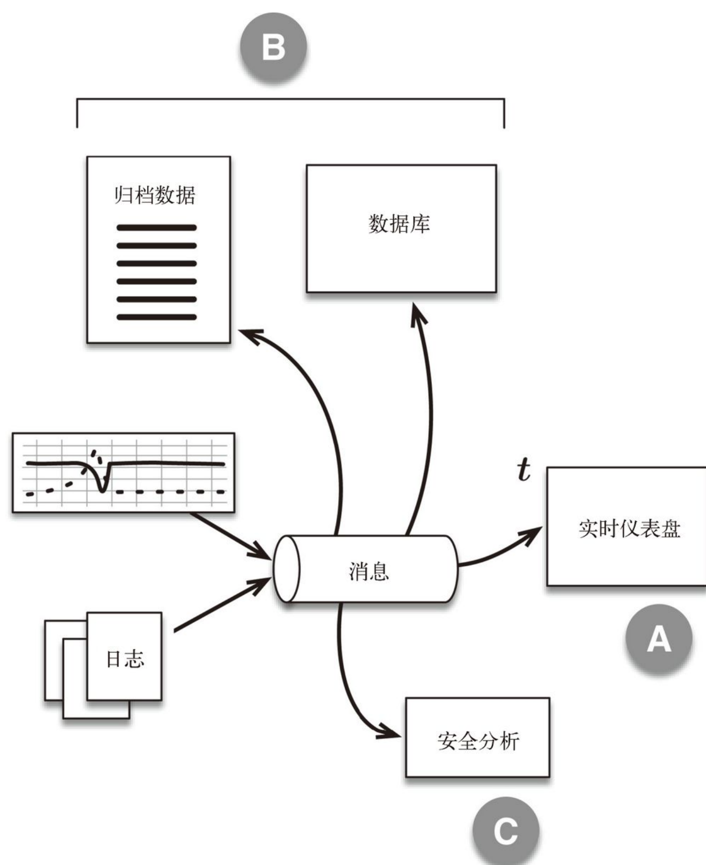


图 **2-5**：流数据消费者并不仅限于实时应用程序，尽管它们是很重要的一种。本图展示了从流处理架构中获益的几类消费者。**A** 组消费者可能做各种实时分析，包括实时更新仪表盘。**B** 组消费者记录数据的当前状态，这些数据可能同时也被存储在数据库或搜索文件中

在其他一些用例中，应用程序使用消息队列中的数据更新本地数据库或者搜索文件（如图 2-5 中的 B 组消费者）。消息队列中的数据往往必须被流处理器聚合或者分析并转换之后，才会输出到数据库中。这是 Flink 擅长的另一个场景。

2.6　流的跨地域复制

流处理架构并不是玩具：它被用于具有重要使命的系统，这些系统要求流处理层和消息传输层具备一些特性。许多关键的业务系统依靠跨

数据中心的一致性，它们不仅需要高效的流处理层，更需要消息传输层拥有可靠的跨地域复制能力。例如，电信公司需要在移动通信基站、用户和处理中心之间共享数据；金融机构需要快速、准确地在相隔很远的办公室之间复制数据，同时控制成本。类似的例子还有很多。

具体来说，数据中心之间的数据复制需要保存消息偏移量，这一点最有用，因为它使得任何数据中心的更新都可以被传播到其他数据中心，且允许双向和循环的数据复制。如果消息偏移量没有被保存，那么另一个数据中心就无法可靠地重启程序。如果不允许其他数据中心更新数据，那么就必须设计可靠的主节点。循环复制则可以避免复制过程出现单点故障。

MapR Streams 目前支持这些功能，但 Kafka 还不支持 1。MapR Streams 的基本原理是，许多流主题被收集在一级数据结构中，该结构也叫作流，它与文件、表格和目录共存于 MapR 的数据平台。这些流成为管理数据复制、生存时间和访问权限的基础。在流中针对主题所做的变更将被贴上源集群的 ID，避免它被无限地循环复制。之后，这些变更被依次传播到其他集群中，并且保留所有的消息偏移量。

> 1作者在此处描述的是 2016 年的情况。至于 Kafka 目前是否支持，请查阅 Kafka 官方文档。——编者注

跨数据中心的流复制能力扩展了流数据和流处理的用途。以在线广告业务为例，流数据分析可以从多个方面提供帮助。结合图 2-5 中的流数据消费者分类，在广告技术领域，实时应用程序（A 组消费者）可能面临实时的广告资源管理，数据库（B 组消费者）当前状态的视图可能是 cookie 档案，流重播（C 组消费者）则可以用来检测虚假点击。

一个挑战是，同一个广告的不同竞价由不同的数据中心处理，但数据都取自同一个广告资源池。在准确性和速度都非常关键的广告技术领域，不同的数据中心如何协调可用的广告资源呢？消息流作为正确的数据源被各个数据中心共享。在本例中，跨数据中心的流复制能力非常有用，MapR Streams 就有这样的能力。

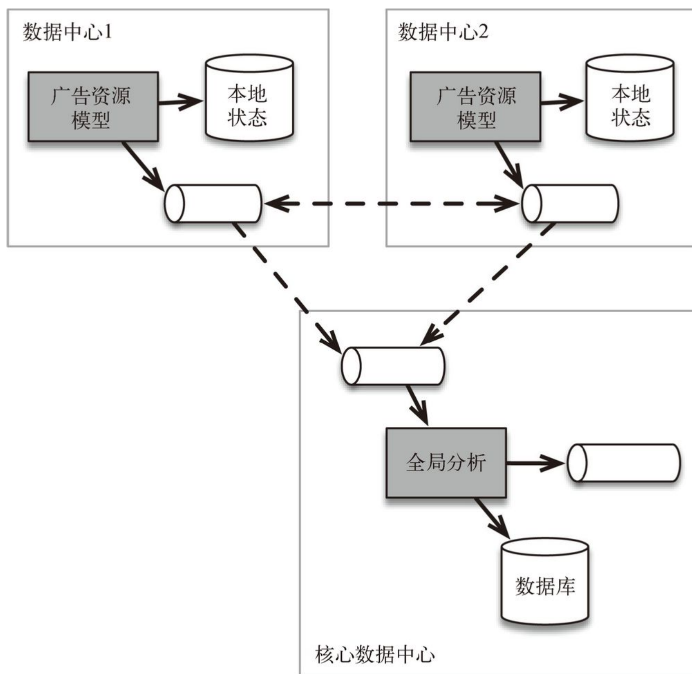


图 **2-6**：来自广告技术领域的例子：流数据分析在不同的数据中心进行，且数据中心有多种基于模型的应用程序，**Flink** 对这些应用程序非常有用。每个本地数据中心都需要保存自己的当前事务状态，而它们都从同一个广告资源池读取数据。另一个需求是与核心数据中心共享数据，并且可以利用 **Flink** 进行全局分析。本用例需要高效且准确地进行跨地域复制

除了能让所有的业务部门了解共享资源的最新状态（这个例子在其他许多行业也同样适用），跨数据中心的流复制能力还有其他优势。拥有不止一个数据中心，可以缓解高吞吐所带来的压力；广告的竞价和投放可以由靠近最终用户的数据中心处理，从而降低网络传输带来的延迟。另外，多个数据中心也可以进行备份，从而防止突发灾难时丢失数据。

我们在第 1 章和第 2 章中看到，流处理的方式符合连续事件发生的自然规律。我们还讨论了流处理架构的诸多好处，这种架构结合了 Kafka 或 MapR Streams 等高效的消息传输技术，以及 Flink 流处理器。

在第 3 章中，我们将探讨 Flink 的关键特性，并在深入学习 Flink 的工作原理之前，概览它的用途。

第 3 章　Flink 的用途

Flink 为流处理器开辟了新的用武之地，它使第 2 章所述的流处理架构变得完整。它的一大优势便是，使应用程序的构建过程符合自然规律。为了了解 Flink 的用途及用法，我们来看一看令它具有多用途的几个核心特点，特别是它如何保障数据的正确性。

3.1　不同类型的正确性

第 1 章介绍了流处理欠佳的后果。本章讨论 Flink 如何正确地进行流处理，以及保障正确性到底意味着什么。人们往往将正确性等同于准确性——以计数为例，计数结果是否“正确”？这个例子很好地诠释了正确性，但是正确性的影响因素很多，当思考计算怎样才能契合需要建模和分析的场景时，更是如此。换句话说，在处理数据时，需要解决这几个问题：“我需要什么？”“我期望什么？”“我在什么时候需要得到结果？”

3.1.1　符合产生数据的自然规律

流处理器（尤其是 Flink）的正确性体现在计算窗口的定义符合数据产生的自然规律。举个例子，通过点击流数据追踪某网站的 3 个访问者（图 3-1 中的 A、B 和 C）的活动。对于每个访问者来说，活动是不连续的。在访问时间段内，事件数据被收集起来；当访问者起身去喝茶或喝咖啡时，或者当他们因为老板从身边经过而切换回工作页面时，数据就产生了间隙。处理框架能够将访问者行为分析的计算窗口与实际的访问时间段吻合到什么程度？换句话说，计算窗口与会话窗口吻合吗？

首先让我们来看看，当访问者行为分析通过微批处理方法或者固定的计算窗口来处理时，会发生什么情况，如图 3-1 所示。

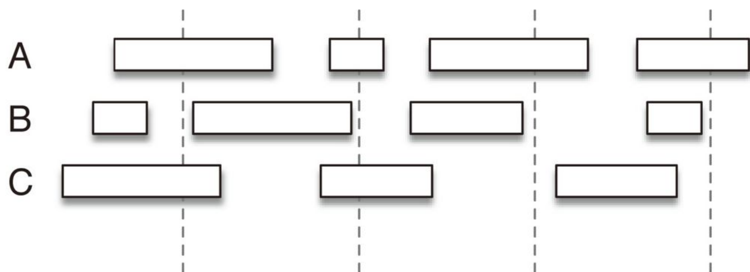


图 **3-1**：采用微批处理方法时，很难使计算窗口（虚线所示）与会话窗口（长方形所示）吻合

由微批处理方法得到的计算窗口是人为设置的，因此很难与会话窗口吻合。使用 Flink 的流处理 API，可以更灵活地定义计算窗口，因此这个问题迎刃而解。举个例子，开发人员可以设置非活动阈值，若超过这个阈值（例如 5 分钟），就可以判断活动结束。图 3-2 展示了这种开窗方式。

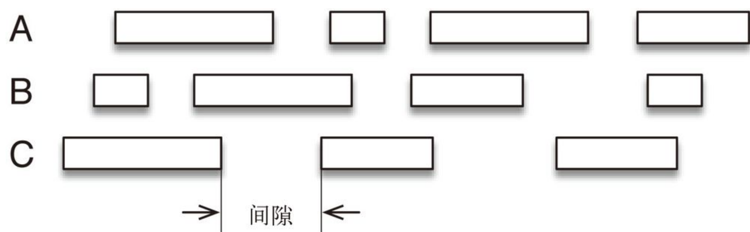


图 **3-2**：**Flink** 的流处理能力能够使计算窗口与会话窗口吻合。如图所示，计算窗口随间隙出现。在本例中，相邻事件之间都有间隙，间隙的时长都超过了预先定义的阈值

Flink 能做到这一点的根本原因是，它可以根据真实情况设置计算窗口。

事件时间与处理时间

> 值得注意的是，时间的指定方式不止一种。在图 3-2 的例子中，为了将事件指定给某一个窗口，程序员很有可能会选择采用事件时间，即事件实际发生的时间。另一种方式是采用处理时间，即事件流数据开始被程序处理的时间。第 4 章将深入介绍 Flink 中的时间概念以及开窗方式。

3.1.2 事件时间

一般而言，流处理架构不常采用事件时间，尽管越来越多的人这样做。Flink 能够完美地做到这一点，这在实现计算的正确性上非常有用。为了获得最佳的计算结果，系统需要能够通过数据找到事件发生的时间，而不是只采用处理时间。

Flink 理解事件时间的这种能力保障了正确性。2016 年，时任 data Artisans 公司应用工程总监的 Jamie Grier 在 OSCON 大会上展示了这一点。他通过生成的数据模拟压力传感器的测量结果，并写了一个 Flink 程序来计算以 1 秒为计算窗口、每秒内正弦波的数值之和。正确的结果是 0。他比较了用处理时间划分窗口和用事件时间划分窗口的差别。采用处理时间时，结果总是或多或少地有些偏差；采用事件时间时，则总是可以获得正确的结果，如图 3-3 所示。

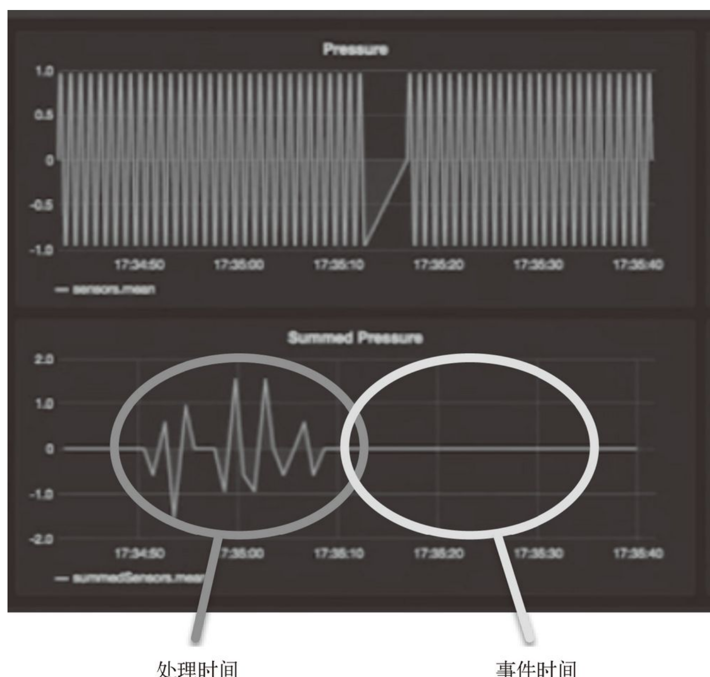


图 **3-3**：从处理时间切换到事件时间，让许多计算工作完成得更好。用处理时间来计算会导致错误，用事件时间则能得到正确的结果（图片来源：**Jamie Grier** 于 **2016** 年 **5** 月在 **OSCON** 大会上所做的演示）

与其他流处理系统相比，Flink 的一个优势就是能区分不同类型的时

间。

3.1.3　发生故障后仍保持准确

若想使计算保持准确，就必须跟踪计算状态。如果计算框架本身不能做到这一点，就必须由应用程序的开发人员来完成这个任务。连续的流处理很难跟踪计算状态，因为计算过程没有终点。实际上，对状态的更新是持续进行的。

Flink 解决了可能影响正确性的几个问题，包括如何在故障发生之后仍能进行有状态的计算。

Flink 所用的技术叫作检查点（checkpoint），第 5 章会详细介绍它的原理。在每个检查点，系统都会记录中间计算状态，从而在故障发生时准确地重置。这一方法使系统以低开销的方式拥有了容错能力——当一切正常时，检查点机制对系统的影响非常小。

值得注意的是，检查点也是 Flink 能够按需重新处理数据的关键所在。毕竟，并不是只有在发生故障之后才会重新处理数据。比如，在运行新模型或者修复 bug 时，就可能需要重播并重新处理事件流数据。Flink 成全了这些操作。

> Flink 的检查点特性在流处理器中是独一无二的，它使得 Flink 可以准确地维持状态，并且高效地重新处理数据。

3.1.4　及时给出所需结果

Flink 能够满足低延迟应用程序的需要，将这算作一种正确性可能出人意料。我们换个角度看看：有些计算结果或许很准确，例如求和或者求平均值的结果，但是如果没有及时地取得结果，那么很难说它们是正确的。举一个例子，假设你在开车上班的途中想通过智能手机上的实时路况查询及导航应用程序选择一条畅通的路，如果应用程序花了 2 小时才把查询结果发给你，那么结果再准确也是无用的。哪怕只有 5 秒钟的延迟也足以造成麻烦，因为你可能已经拐错了弯。

可见，在某些情况下，极低的延迟非常重要，它决定了系统能够及时地给出所需结果，而不仅仅是完成计算。Flink 的实时且容错的流处理能力可以满足这类需求。

3.1.5　使开发和运维更轻松

Flink 与用户交互的接口也有助于保障正确性。完备的语义简化了开发工作，进而降低了出错率。此外，Flink 还承担了跟踪计算状态的任务，从而减轻了开发人员的负担，简化了编程工作，并提高了应用程序的成功率。用同一种技术来实现流处理和批处理，大大地简化了开发和运维工作。

3.2 分阶段采用Flink

尽管 Flink 拥有非常丰富的功能，并能处理极为复杂的数据，但是没有必要为了采用 Flink 而彻底抛弃其他技术。流处理架构可以分步来实现。有些公司在引入流处理架构时，先实现简单的应用程序，等到熟悉后再推广。虽然试点应用程序的类型完全取决于公司的需求，但是许多公司都有相似的流处理“价值链”。

接下来，让我们抓住机会深入了解 Flink 能做什么以及它是如何做到的。第 4~6 章将介绍 Flink 的一些基本功能。

第 4 章 对时间的处理

用流处理器编程和用批处理器编程最关键的区别在于对时间的处理。举一个非常简单的例子：计数。事件流数据（如微博内容、点击数据和交易数据）不断产生，我们需要用 key 将事件分组，并且每隔一段时间（比如一小时）就针对每一个 key 对应的事件计数。这是众所周知的“大数据”应用，与 MapReduce 的词频统计例子相似。

4.1 采用批处理架构和Lambda架构计数

尽管看起来简单，但是大规模的计数任务在实践中出人意料地困难。当然，计数无处不在。针对联机分析处理多维数据集的聚合或其他操作，都可以简单地归结为计数。图 4-1 展示了如何采用传统的批处理架构实现计数任务。

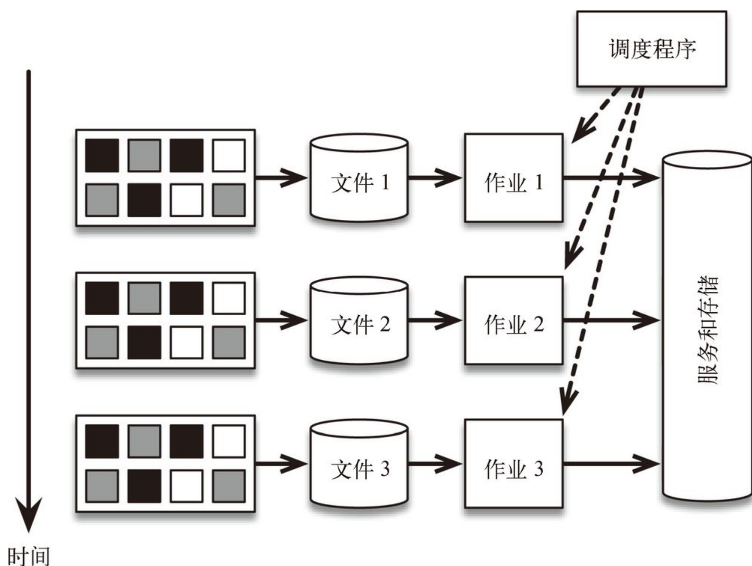


图 **4-1**：用定期运行的批处理作业来实现应用程序的持续性。数据被持续地分割为文件（如以一小时为单位）；然后，批处理作业将文件作为输入，以此达到持续处理数据的效果

在该架构中，持续摄取数据的管道每小时创建一次文件。这些文件通常被存储在 HDFS 或 MapR-FS 等分布式文件系统中。像 Apache Flume 这样的工具可以用于完成上述工作。由调度程序安排批处理作业（如 MapReduce 作业）分析最近生成的一个文件（将文件中的事件按 key 分组，计算每个 key 对应的事件数），然后输出计数结果。对于每个使用 Hadoop 的公司来说，其集群都有多个类似的管道。

这种架构完全可行，但是存在以下问题。

- 太多独立的部分。为了计算数据中的事件数，这种架构动用了太多系统。每一个系统都有学习成本和管理成本，还可能存在 bug。
- 对时间的处理方法不明确。假设需要改为每 30 分钟计数一次。这个变动涉及工作流调度逻辑（而不是应用程序代码逻辑），从而使 DevOps 问题与业务需求混淆。
- 预警。假设除了每小时计数一次外，还需要尽可能早地收到计数预警（比如在事件数超过 10 时预警）。为了做到这一点，可以在定期运行的批处理作业之外，引入 Storm 来采集消息流（Kafka 或者 MapR Streams）。Storm 实时提供近似的计数，

批处理作业每小时提供准确的计数。但是这样一来，就向架构增加了一个系统，以及与之相关的新编程模型。上述架构叫作 Lambda 架构，如图 4-2 所示。第 1 章有过简要的介绍。

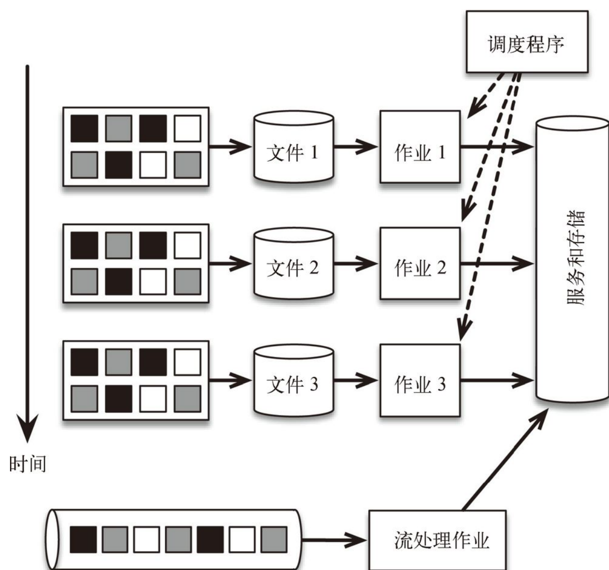


图 **4-2**：**Lambda** 架构用定期运行的批处理作业来实现应用程序的持续性，并通过流处理器获得预警。流处理器实时提供近似结果；批处理层最终会对近似结果予以纠正

- 乱序事件流。在现实世界中，大多数事件流都是乱序的，即事件的实际发生顺序（事件数据在生成时被附上时间戳，如智能手机记录下用户登录应用程序的时间）和数据中心所记录的顺序不一样。这意味着本属于前一批的事件可能被错误地归入当前一批。批处理架构很难解决这个问题，大部分人则选择忽视它。

- 批处理作业的界限不清晰。在该架构中，“每小时”的定义含糊不清，分割时间点实际上取决于不同系统之间的交互。充其量也只能做到大约每小时分割一次，而在分割时间点前后的事件既可能被归入前一批，也可能被归入当前一批。将数据流以小时为单位进行分割，实际上是最简单的方法。假设需要根据产生数据的时间段（如从用户登录到退出）生成聚合结果，而不是简单地以小时为单位分割数据，则用如图 4-1 和图 4-2 所示的架构无法直接满足需求。

4.2 采用流处理架构计数

当然有更好的办法来对事件流进行计数。如你所想，计数是流处理用例，上一节只是使用定期运行的批处理作业来模拟流处理。此外，必须把各种系统耦合在一起。图 4-3 展示了采用流处理架构的应用程序模型。

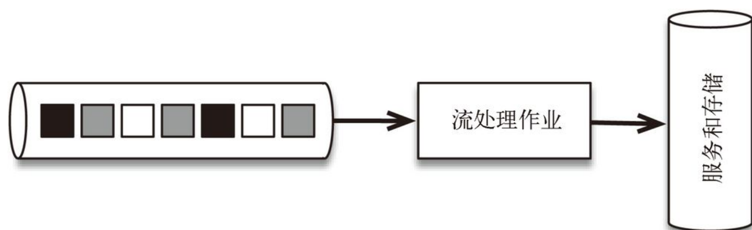


图 **4-3**：通过流处理架构实现应用程序的持续性。水平圆柱体表示消息传输系统（**Kafka** 或 **MapR Streams**）。消息传输系统为负责处理所有数据的流处理器（在本例中是 **Flink**）提供流数据，产生的结果既是实时的，也是正确的

事件流由消息传输系统提供，并且只被单一的 Flink 作业处理，从而以小时为单位计数和预警（后者可选）。这种方法直接解决了上一节提到的所有问题。Flink 作业的速度减慢或者吞吐量激增只会导致事件在消息传输系统中堆积。以时间为单位把事件流分割为一批批任务（称作窗口），这种逻辑完全嵌入在 Flink 程序的应用逻辑中。预警由同一个程序生成，乱序事件由 Flink 自行处理。要从以固定时间分组改为根据产生数据的时间段分组，只需在 Flink 程序中修改对窗口的定义即可。此外，如果应用程序的代码有过改动，只需重播 Kafka 主题，即可重播应用程序。采用流处理架构，可以大幅减少需要学习、管理和编写代码的系统。Flink 应用程序用来计数的代码非常简单，如下所示。

```
DataStream<LogEvent> stream = env
  // 通过Kafka生成数据流
  .addSource(new FlinkKafkaConsumer(...))
  // 分组
  .keyBy("country")
  // 将时间窗口设为60分钟
  .timeWindow(Time.minutes(60))
  // 针对每个时间窗口进行操作
  .apply(new CountPerWindowFunction());
```

流处理区别于批处理最主要的两点是：流即是流，不必人为地将它分割为文件；时间的定义被明确地写入应用程序代码（如以上代码的时间窗口），而不是与摄取、计算和调度等过程牵扯不清。

流处理系统中的批处理

第 1 章讨论过微批处理，它是介于流处理和批处理之间的方法。实际上，微批处理的含义取决于具体情况。从某种角度来说，图 4-1 中的批处理架构也可以称为微批处理架构，前提是文件足够小。

Storm Trident 是这样实现微批处理的：它先创建一个大的 Storm 事件，其中包含固定数量的子事件；然后将这些聚合在一起的子事件用持续运行的 Storm 拓扑处理。Spark Streaming 的微批处理架构和批处理架构本质上是一致的，只不过对用户隐藏了前两步（摄取和存储），并将每份微批数据用预写日志（而不是文件）的形式存储在内存中。包括 Flink 在内的所有现代流处理器在内部都使用了某种形式的微批处理技术，在 shuffle 阶段将含有多个事件的缓冲容器通过网络发送，而不是发送单个事件。这些形式各不相同。

需要说明的是，流处理系统中的批处理必须符合以下两点要求。

- 批处理只作为提高系统性能的机制。批量越大，系统的吞吐量就越大。
- 为了提高性能而使用的批处理必须完全独立于定义窗口时所用的缓冲，或者为了保证容错性而提交的代码，也不能作为 API 的一部分。否则，系统将受到限制，并且变得脆弱且难以使用。

应用程序开发人员和数据处理系统的用户不需要考虑系统是否可以执行微批处理以及如何执行，而是需要考虑系统是否可以处理乱序事件流以及不一致的窗口，是否可以在提供准确的聚合结果之外还提供预警，以及是否可以准确地重播历史数据；还需要考虑系统的性能特征（低延迟和高吞吐），以及如何在发生故障时保证系统持续运行。

4.3　时间概念

在流处理中，主要有两个时间概念 1。

> 1本章涉及的许多想法都是由 Google Dataflow 团队（即现在的 Apache Beam 团队）提出的。团队成员包括 Tyler Akidau 和 Frances Perry 等人。如果想了解更多关于 Dataflow 模型的内容，请参考 Tyler Akidau 的文章 *Streaming 101* 和 *Streaming 102*。Flink 处理时

间和窗口的机制很大程度上来源于 Tyler Akidau 等人所著的关于 Dataflow 模型的论文。

- 事件时间，即事件实际发生的时间。更准确地说，每一个事件都有一个与它相关的时间戳，并且时间戳是数据记录的一部分（比如手机或者服务器的记录）。事件时间其实就是时间戳。
- 处理时间，即事件被处理的时间。处理时间其实就是处理事件的机器所测量的时间。

图 4-4 说明了事件时间和处理时间的区别。

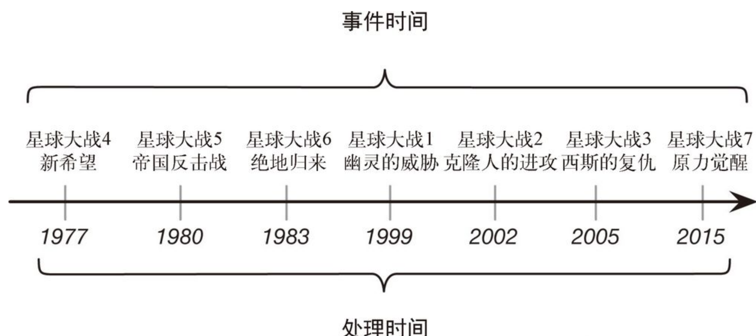


图 **4-4**：事件时间顺序与处理时间顺序不一致的乱序事件流

以《星球大战》系列电影为例。首先上映的 3 部电影是该系列中的第 4、5、6 部（这是事件时间），它们的上映年份分别是 1977 年、1980 年和 1983 年（这是处理时间）。之后按事件时间上映的第 1、2、3、7 部，对应的处理时间分别是 1999 年、2002 年、2005 年和 2015 年。由此可见，事件流的顺序可能是乱的（尽管年份顺序一般不会乱）。

通常还有第 3 个时间概念，即摄取时间，也叫作进入时间。它指的是事件进入流处理框架的时间。缺乏真实事件时间的数据会被流处理器附上时间戳，即流处理器第一次看到它的时间（这个操作由 source 函数完成，它是程序的第一个处理节点）。

在现实世界中，许多因素（如连接暂时中断，不同原因导致的网络延迟，分布式系统中的时钟不同步，数据速率陡增，物理原因，或者运气差）使得事件时间和处理时间存在偏差（即事件时间偏差）。事件时间顺序和处理时间顺序通常不一致，这意味着事件以乱序到达流处理器。

根据应用程序的不同，两个时间概念都很有用。有些应用程序（如一

些预警应用程序）需要尽可能快地得到结果，即使有小的误差也没关系。它们不必等待迟到的事件，因此适合采用处理时间语义。其他一些应用程序（如欺诈检测系统或者账单系统）则对准确性有要求：只有在时间窗口内发生的事件才能被算进来。对于这些应用程序来说，事件时间语义才是正确的选择。也有两者都采用的情况，比如既要准确地计数，又要提供异常预警。

> Flink 允许用户根据所需的语义和对准确性的要求选择采用事件时间、处理时间或摄取时间定义窗口。

当采用事件时间定义窗口时，应用程序可以处理乱序事件流以及变化的事件时间偏差，并根据事件实际发生的时间计算出有意义的结果。

4.4 窗口

4.1 节举例说明了如何在 Flink 中定义时间窗口并以小时为单位生成聚合结果。窗口是一种机制，它用于将许多事件按照时间或者其他特征分组，从而将每一组作为整体进行分析（比如求和）。

4.4.1 时间窗口

时间窗口是最简单和最有用的一种窗口。它支持滚动和滑动。举一个例子，假设要对传感器输出的数值求和。

一分钟滚动窗口收集最近一分钟的数值，并在一分钟结束时输出总和，如图 4-5 所示。

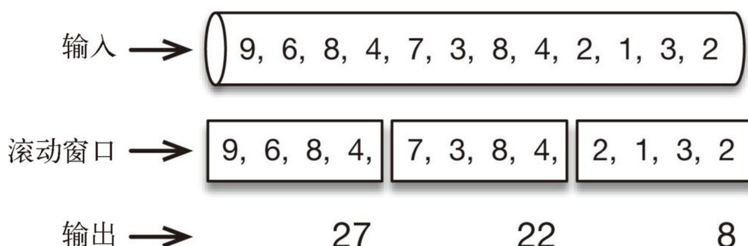


图 **4-5**：一分钟滚动窗口计算最近一分钟的数值总和

一分钟滑动窗口计算最近一分钟的数值总和，但每半分钟滑动一次并输出结果，如图 4-6 所示。

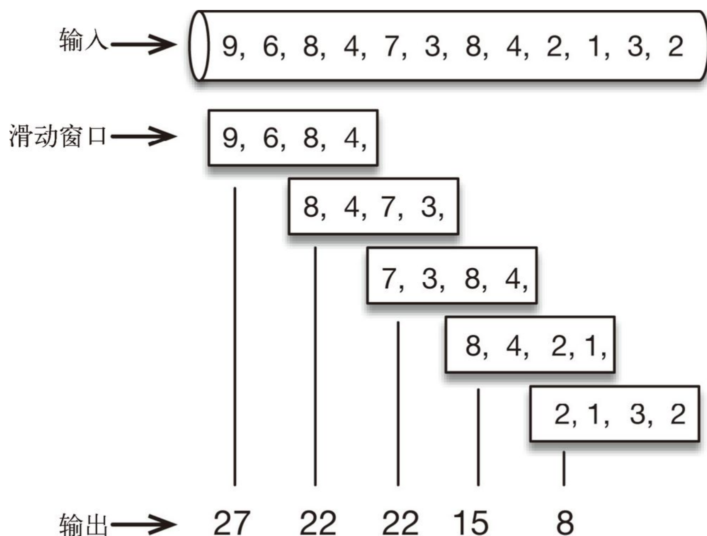


图 **4-6**：一分钟滑动窗口每半分钟计算一次最近一分钟的数值总和

第一个滑动窗口对 9、6、8 和 4 求和，得到 27。半分钟后，窗口滑动，然后对 8、4、7 和 3 求和，得到 22，照此类推。

在 Flink 中，一分钟滚动窗口的定义如下。

```
stream.timeWindow(Time.minutes(1))
```

每半分钟（即 30 秒）滑动一次的一分钟滑动窗口如下所示。

```
stream.timeWindow(Time.minutes(1), Time.seconds(30))
```

4.4.2　计数窗口

Flink 支持的另一种常见窗口叫作计数窗口。采用计数窗口时，分组依据不再是时间戳，而是元素的数量。例如，图 4-6 中的滑动窗口也可以解释为由 4 个元素组成的计数窗口，并且每两个元素滑动一次。滚动和滑动的计数窗口分别定义如下。

```
stream.countWindow(4)
stream.countWindow(4, 2)
```

虽然计数窗口有用，但是其定义不如时间窗口严谨，因此要谨慎使用。时间不会停止，而且时间窗口总会“关闭”。但就计数窗口而言，假设其定义的元素数量为 100，而某个 key 对应的元素永远达不到 100 个，那么窗口就永远不会关闭，被该窗口占用的内存也就浪费了。一种解决办法是用时间窗口来触发超时，4.4.4 节会详细介绍。

4.4.3　会话窗口

Flink 支持的另一种很有用的窗口是会话窗口。第 3 章提到过这个概念。会话指的是活动阶段，其前后都是非活动阶段，例如用户与网站进行一系列交互（活动阶段）之后，关闭浏览器或者不再交互（非活动阶段）。会话需要有自己的处理机制，因为它们通常没有固定的持续时间（有些 30 秒就结束了，有些则长达一小时），或者没有固定的交互次数（有些可能是 3 次点击后购买，另一些可能是 40 次点击却没有购买）。

 Flink 是目前唯一 **2** 支持会话窗口的开源流处理器。

> 2作者在此处描述的是 2016 年的情况。目前，Apache Beam 和 Apache Spark 也支持会话窗口。——译者注

在 Flink 中，会话窗口由超时时间设定，即希望等待多久才认为会话已经结束。举例来说，以下代码表示，如果用户处于非活动状态长达 5 分钟，则认为会话结束。

```
stream.window(SessionWindows.withGap(Time.minutes(5))
```

4.4.4　触发器

除了窗口之外，Flink 还提供触发机制。触发器控制生成结果的时间，即何时聚合窗口内容并将结果返回给用户。每一个默认窗口都有一个触发器。例如，采用事件时间的时间窗口将在收到水印时被触发。对于用户来说，除了收到水印时生成完整、准确的结果之外，也可以实现自定义的触发器（例如每秒提供一次近似结果）。

4.4.5　窗口的实现

在 Flink 内部，所有类型的窗口都由同一种机制实现。虽然实现细节

对于普通用户来说并不重要，但是仍然需要注意以下两点。

- 开窗机制与检查点机制（第 5 章将详细讨论）完全分离。这意味着窗口时长不依赖于检查点间隔。事实上，窗口完全可以没有“时长”（比如上文中的计数窗口和会话窗口的例子）。
- 高级用户可以直接用基本的开窗机制定义更复杂的窗口形式（如某种时间窗口，它可以基于计数结果或某一条记录的值生成中间结果）。

4.5　时空穿梭

流处理架构的一个核心能力是时空穿梭。如果所有的数据处理工作都由流处理器完成，那么应用程序如何演进呢？我们如何处理历史数据，又如何重新处理数据呢？（假设出于调试或者审计的目的，需要重新处理数据。）

如图 4-7 所示，时空穿梭意味着将数据流倒回至过去的某个时间，重新启动处理程序，直到处理至当前时间为止。像 Kafka 和 MapR Streams 这样的现代传输层，支持时空穿梭，这使得它们与更早的解决方案有所区别。实时流处理总是在处理最近的数据（即图中“当前时间”的数据），历史流处理则从过去开始，并且可以一直处理至当前时间。

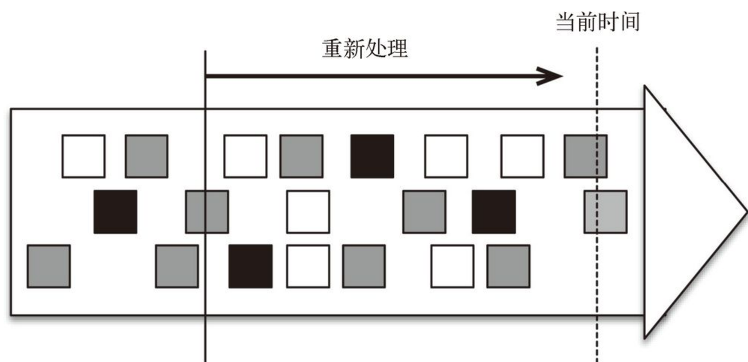


图 **4-7**：流处理架构拥有时空穿梭（即重新处理数据）的能力。流处理器支持事件时间，这意味着将数据流“倒带”，用同一组数据重新运行同样的程序，会得到相同的结果

若要按时间回溯并正确地重新处理数据，流处理器必须支持事件时间。

如果窗口的设定是根据系统时间而不是时间戳，那么每次运行同样的程序，都会得到不同的结果。事件时间使数据处理结果具有确定性，因为用同一组数据运行同样的程序，会得到相同的结果。

4.6 水印

支持事件时间对于流处理架构而言至关重要，因为事件时间能保证结果正确，并使流处理架构拥有重新处理数据的能力。当计算基于事件时间时，如何判断所有事件是否都到达，以及何时计算和输出窗口的结果呢？换言之，如何追踪事件时间，并知晓输入数据已经流到某个事件时间了呢？为了追踪事件时间，需要依靠由数据驱动的时钟，而不是系统时钟。

以图 4-5 中的一分钟滚动窗口为例。假设第一个窗口从 10:00:00 开始（即从 10 时 0 分 0 秒开始），需要计算从 10:00:00 到 10:01:00 的数值总和。当时间就是记录的一部分时，我们怎么知道 10:01:00 已到呢？换句话说，我们怎么知道盖有时间戳 10:00:59 的元素还没到呢？

Flink 通过水印来推进事件时间。水印是嵌在流中的常规记录，计算程序通过水印获知某个时间点已到。对于上述一分钟滚动窗口，假设水印标记时间为 10:01:00（或者其他时间，如 10:03:43），那么收到水印的窗口就知道不会再有早于该时间的记录出现，因为所有时间戳小于或等于该时间的事件都已经到达。这时，窗口可以安全地计算并给出结果（总和）。水印使事件时间与处理时间完全无关。迟到的水印（“迟到”是从处理时间的角度而言）并不会影响结果的正确性，而只会影响收到结果的速度。

水印是如何生成的

在 Flink 中，水印由应用程序开发人员生成，这通常需要对相应的领域有一定的了解。完美的水印永远不会错：时间戳小于水印标记时间的事件不会再出现。在特殊情况下（例如非乱序事件流），最近一次事件的时间戳就可能是完美的水印。启发式水印则相反，它只估计时间，因此有可能出错，即迟到的事件（其时间戳小于水印标记时间）晚于水印出现。针对启发式水印，Flink 提供了处理迟到元素的机制。

设定水印通常需要用到领域知识。举例来说，如果知道事件的迟到时间不会超过 5 秒，就可以将水印标记时间设为收到的最大时间戳减去

5 秒。另一种做法是，采用一个 Flink 作业监控事件流，学习事件的迟到规律，并以此构建水印生成模型。

> 水印为以事件时间说明输入数据的完整性提供了一种机制，这种机制可能是启发式的。

如果水印迟到得太久，收到结果的速度可能就会很慢，解决办法是在水印到达之前输出近似结果（Flink 可以实现）。如果水印到达得太早，则可能收到错误结果，不过 Flink 处理迟到数据的机制可以解决这个问题。上述问题看起来很复杂，但是恰恰符合现实世界的规律——大部分真实的事件流都是乱序的，并且通常无法了解它们的乱序程度（因为理论上不能预见未来）。水印是唯一让我们直面乱序事件流并保证正确性的机制；否则只能选择忽视事实，假装错误的结果是正确的。

4.7　真实案例：爱立信公司的Kappa架构

考虑到由爱立信公司提供技术支持的运营商通常拥有庞大的数据规模（每天处理 10TB~100TB 的数据，每秒处理 10 万~100 万个事件），该公司的一支团队决定实现所谓的 **Kappa** 架构 3。2014 年，Kafka 的创始人之一 Jay Kreps 为 O'Reilly Radar 撰写了一篇批评 Lambda 架构的文章 4，并在其中开玩笑式地创造了“Kappa 架构”这个词。其实，Kappa 架构正是第 2 章所讨论的流处理架构。其中，数据流是设计核心；数据源不可变更；架构采用像 Flink 这样的单一流分析框架处理新鲜数据，并通过流重播处理历史数据。

> 3本节内容基于 Nicolas Seyvet 和 Ignacio Mulas Viela 在 Strata + Hadoop World London 2016 研讨会上所做的演讲。Ignacio Mulas Viela 在 2015 年的 Flink Forward 研讨会上也做过相关的演讲。

> 4https://www.oreilly.com/ideas/questioning-the-lambda-architecture

爱立信公司需要实时分析云基础设施的系统性能指标和日志，从而持续地监视系统行为，以确定是一切正常，还是有“新奇点”出现。“新奇点”既可能是异常行为，也可能是系统状态变更，例如加入了新虚拟机。爱立信团队使用的方法是将一个贝叶斯在线学习模型应用于包含电信云监控系统多个指标的数据流（遥测信息和日志事件）。爱立信公司的研究人员 Nicolas Seyvet 和 Ignacio Mulas Viela 说道：

> 该架构在不断地适应（学习）新系统常态的同时，能

够快速且准确地发现异常。这使它成为理想工具，并能够极大地降低因大型计算设施运行而产生的维护成本。

图 4-8 展示了爱立信团队构建的数据管道。

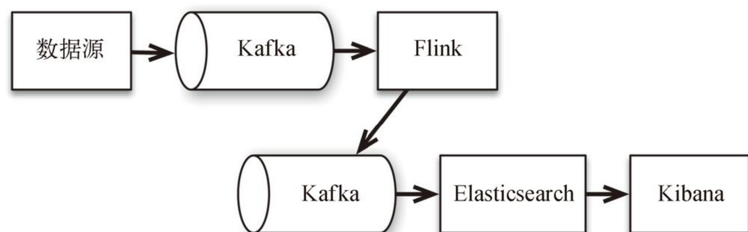


图 **4-8**：爱立信团队采用的基于 **Flink** 的流处理架构

推送给 Kafka 的原始数据是来自云基础设施中的所有实体机和虚拟机的遥测信息和日志事件。它们经过不同的 Flink 作业消费之后，被写回 Kafka 主题里，然后再从 Kafka 主题里被推送给搜索引擎 Elasticsearch 和可视化系统 Kibana。这种架构让每个 Flink 作业所执行的任务有清晰的定义，一个作业的输出可以成为另一个作业的输入。图 4-9 展示了异常检测管道，每个中间流都是 Kafka 主题（以分配给它的数据命名），每个长方形代表一个 Flink 作业。

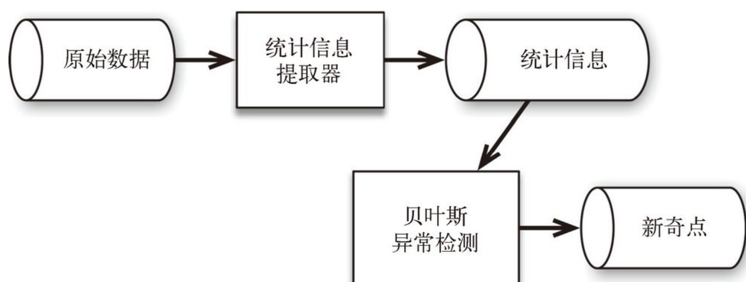


图 **4-9**：爱立信公司的异常检测管道采用 **Flink** 实现统计信息提取和异常检测

在本案例中，为什么 Flink 对事件时间的支持很重要呢？有两个原因。

(1) 有助于准确地识别异常。时间对识别异常很重要。当许多日志事件在同一时间出现时，通常说明可能有错误发生。为了将这些事件正

确地分组和归类，考虑它们的真实时间（而不是处理时间）很重要。

(2) 有助于采用流处理架构。在流处理架构中，所有的计算都由流处理器完成。升级应用程序的做法是将它们在流处理器中再次执行。用同一种计算运行两次同样的数据，必须得到同样的结果，这只有依靠事件时间操作才能实现。

第 5 章　有状态的计算

流式计算分为无状态和有状态两种情况。无状态的计算观察每个独立事件，并根据最后一个事件输出结果。例如，流处理应用程序从传感器接收温度读数，并在温度超过 90 度时发出警告。有状态的计算则会基于多个事件输出结果。以下是一些例子。

- 第 4 章讨论的所有类型的窗口。例如，计算过去一小时的平均温度，就是有状态的计算。
- 所有用于复杂事件处理的状态机。例如，若在一分钟内收到两个相差 20 度以上的温度读数，则发出警告，这是有状态的计算。
- 流与流之间的所有关联操作，以及流与静态表或动态表之间的关联操作，都是有状态的计算。

图 5-1 展示了无状态流处理和有状态流处理的主要区别。无状态流处理分别接收每条记录（图中的黑条），然后根据最新输入的记录生成输出记录（白条）。有状态流处理会维护状态（根据每条输入记录进行更新），并基于最新输入的记录和当前的状态值生成输出记录（灰条）。

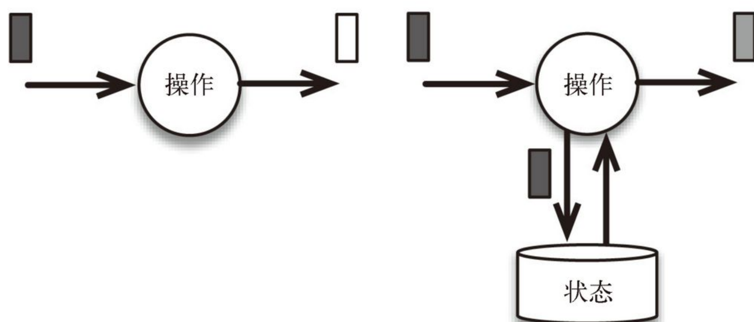


图 **5-1**：无状态流处理与有状态流处理的区别。输入记录由黑条表示。无状态流处理每次只转换一条输入记录，并且仅根据最新的输入记录输出结果（白条）。有状态流处理维护所有已处理记录的状态值，并根据每条新输入的记录更新状态，因此输出记录（灰条）反映的是综合考虑多个事件之后的结果

尽管无状态的计算很重要，但是流处理对有状态的计算更感兴趣。事实上，正确地实现有状态的计算比实现无状态的计算难得多。旧的流处理系统并不支持有状态的计算，而新一代的流处理系统则将状态及其正确性视为重中之重。

5.1　一致性

当在分布式系统中引入状态时，自然也引入了一致性问题。一致性实际上是“正确性级别”的另一种说法，即在成功处理故障并恢复之后得到的结果，与没有发生任何故障时得到的结果相比，前者有多正确？举例来说，假设要对最近一小时登录的用户计数。在系统经历故障之后，计数结果是多少？在流处理中，一致性分为 3 个级别。

- at-most-once：这其实是没有正确性保障的委婉说法——故障发生之后，计数结果可能丢失。
- at-least-once：这表示计数结果可能大于正确值，但绝不会小于正确值。也就是说，计数程序在发生故障后可能多算，但是绝不会少算。
- exactly-once：这指的是系统保证在发生故障后得到的计数结果与正确值一致。

曾经，at-least-once 非常流行。第一代流处理器（如 Storm 和 Samza）刚问世时只保证 at-least-once，原因有二。

(1) 保证 exactly-once 的系统实现起来更复杂。这在基础架构层（决定什么代表正确，以及 exactly-once 的范围是什么）和实现层都很有挑战性。

(2) 流处理系统的早期用户愿意接受框架的局限性，并在应用层想办法弥补（例如使应用程序具有幂等性，或者用批量计算层再做一遍计算）。

最先保证 exactly-once 的系统（Storm Trident 和 Spark Streaming）在性能和表现力这两个方面付出了很大的代价。为了保证 exactly-once，这些系统无法单独地对每条记录运用应用逻辑，而是同时处理多条（一批）记录，保证对每一批的处理要么全部成功，要么全部失败。这就导致在得到结果前，必须等待一批记录处理结束。因此，用户经常不得不使用两个流处理框架（一个用来保证 exactly-once，另一个用来对每个元素做低延迟处理），结果使基础设施更加复杂。曾经，用户不得不在保证 exactly-once 与获得低延迟和效率之间权衡利弊。Flink 避免了这种权衡。

> Flink 的一个重大价值在于，它既保证了 exactly-once，也具有低延迟和高吞吐的处理能力。

从根本上说，Flink 通过使自身满足所有需求来避免权衡，它是业界的一次意义重大的技术飞跃。尽管这在外行看来很神奇，但是一旦了解，就会恍然大悟。

5.2 检查点：保证exactly-once

Flink 如何保证 exactly-once 呢？它使用一种被称为“检查点”的特性，在出现故障时将系统重置回正确状态。下面通过简单的类比来解释检查点的作用。

假设你和两位朋友正在数项链上有多少颗珠子，如图 5-2 所示。你捏住珠子，边数边拨，每拨过一颗珠子就给总数加一。你的朋友也这样数他们手中的珠子。当你分神忘记数到哪里时，怎么办呢？如果项链上有很多珠子，你显然不想从头再数一遍，尤其是当三人的速度不一样却又试图合作的时候，更是如此（比如想记录前一分钟三人一共数了多少颗珠子，回想一下第 4 章中的一分钟滚动窗口）。



图 **5-2**：数环状项链上的珠子看上去毫无意义（甚至有些徒劳无功，因为可以永不停歇地计数），但是它可以用来很好地类比处理永不结束的事件流。在某些文化中，人们仍旧将数珠子视作消磨时间的好方法

于是，你想了一个更好的办法：在项链上每隔一段就松松地系上一根有色皮筋，将珠子分隔开；当珠子被拨动的时候，皮筋也可以被拨动；然后，你安排一个助手，让他在你和朋友拨到皮筋时记录总数。用这种方法，当有人数错时，就不必从头开始数。相反，你向其他人发出错误警示，然后你们都从上一根皮筋处开始重数，助手则会告诉每个人重数时的起始数值，例如在粉色皮筋处的数值是多少。

Flink 检查点的作用就类似于皮筋标记。数珠子这个类比的关键点是：对于指定的皮筋而言，珠子的相对位置是确定的；这让皮筋成为重新计数的参考点。总状态（珠子的总数）在每颗珠子被拨动之后更新一次，助手则会保存与每根皮筋对应的检查点状态，如当遇到粉色皮筋时一共数了多少珠子，当遇到橙色皮筋时又是多少。当问题出现时，这种方法使得重新计数变得简单。

> 检查点是 Flink 最有价值的创新之一，因为它使 Flink 可以保证 exactly-once，并且不需要牺牲性能。

Flink 检查点的核心作用是确保状态正确，即使遇到程序中断，也要正确。记住这一基本点之后，我们用一个例子来看检查点是如何运行的。Flink 为用户提供了用来定义状态的工具。例如，以下这个 Scala

程序按照输入记录的第一个字段（一个字符串）进行分组并维护第二个字段的计数状态。

```
val stream: DataStream[(String, Int)] = ...

val counts: DataStream[(String, Int)] = stream
  .keyBy(record => record._1)
  .mapWithState((in: (String, Int), count: Option[Int]) =>
    count match {
      case Some(c) => ( (in._1, c + in._2), Some(c + in._2) )
      case None => ( (in._1, in._2), Some(in._2) )
    })
```

该程序有两个算子：keyBy 算子用来将记录按照第一个元素（一个字符串）进行分组，根据该 key 将数据进行重新分区，然后将记录再发送给下一个算子：有状态的 map 算子（mapWithState）。map 算子在接收到每个元素后，将输入记录的第二个字段的数据加到现有总数中，再将更新过的元素发射出去。图 5-3 表示程序的初始状态：输入流中的 6 条记录被检查点屏障（checkpoint barrier）隔开，所有的 map 算子状态均为 0（计数还未开始）。所有 key 为 a 的记录将被顶层的 map 算子处理，所有 key 为 b 的记录将被中间层的 map 算子处理，所有 key 为 c 的记录则将被底层的 map 算子处理。

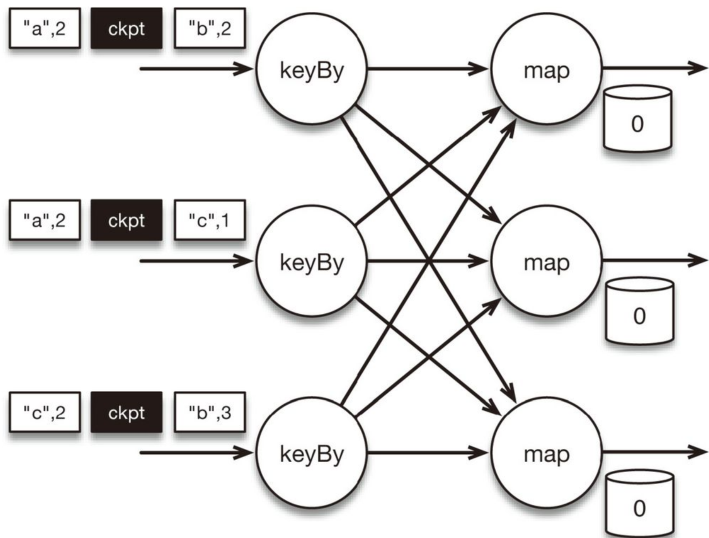


图 5-3：程序的初始状态。注意，**a**、**b**、**c** 三组的初始计数状态都是 **0**，即三个圆柱上的值。**ckpt** 表示检查点屏障。每条记录在处理顺序

上严格地遵守在检查点之前或之后的规定，例如 **["b",2]** 在检查点之前被处理，**["a",2]** 则在检查点之后被处理

当该程序处理输入流中的 6 条记录时，涉及的操作遍布 3 个并行实例（节点、CPU 内核等）。那么，检查点该如何保证 exactly-once 呢？

检查点屏障和普通记录类似。它们由算子处理，但并不参与计算，而是会触发与检查点相关的行为。当读取输入流的数据源（在本例中与 `keyBy` 算子内联）遇到检查点屏障时，它将其在输入流中的位置保存到稳定存储中。如果输入流来自消息传输系统（Kafka 或 MapR Streams），这个位置就是偏移量。Flink 的存储机制是插件化的，稳定存储可以是分布式文件系统，如 HDFS、S3 或 MapR-FS。图 5-4 展示了这个过程。

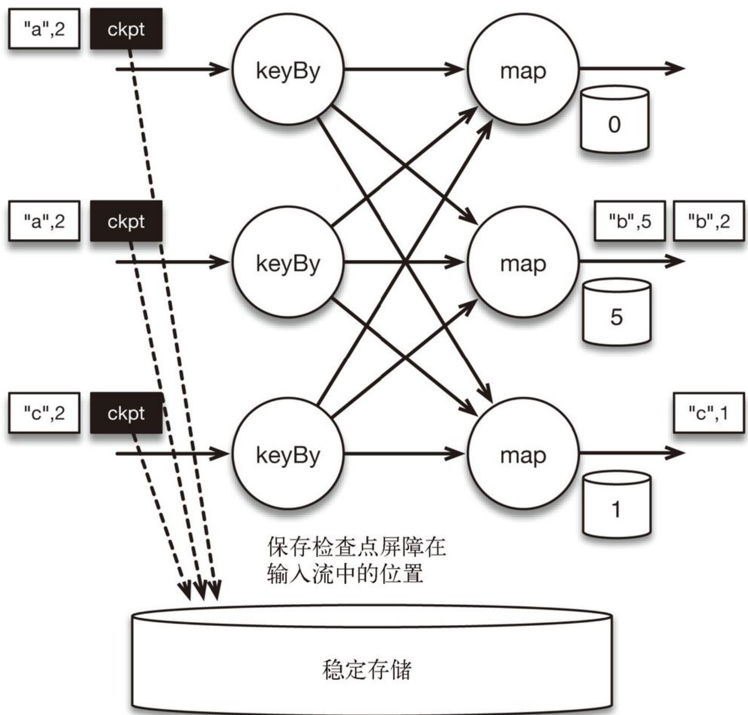


图 5-4：当 Flink 数据源（在本例中与 `keyBy` 算子内联）遇到检查点屏障时，它会将其在输入流中的位置保存到稳定存储中。这让 Flink 可以根据该位置重启输入

检查点屏障像普通记录一样在算子之间流动。当 `map` 算子处理完前

3 条记录并收到检查点屏障时，它们会将状态以异步的方式写入稳定存储，如图 5-5 所示。

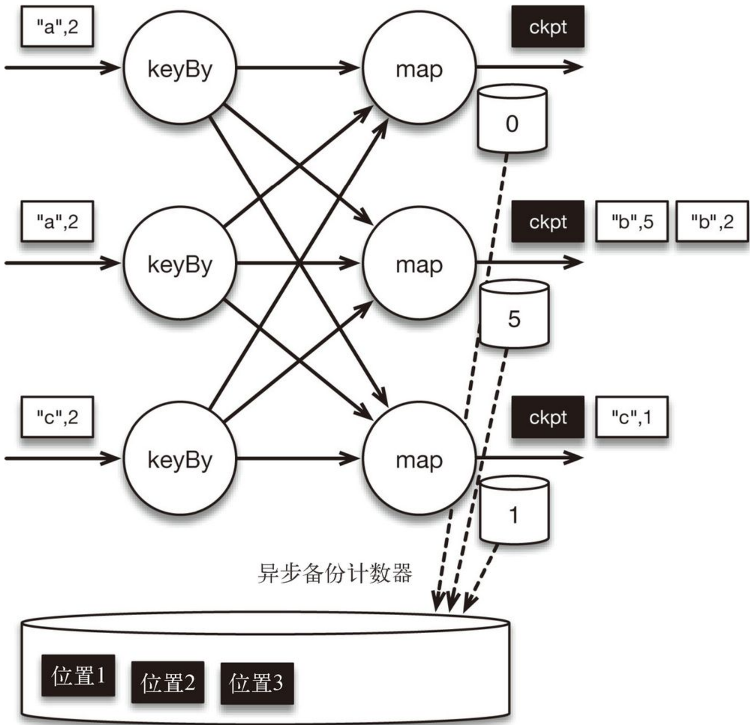


图 **5-5**：位于检查点之前的所有记录（**["b",2]**、**["b",3]** 和 **["c",1]**）被 **map** 算子处理之后的情况。此时，稳定存储已经备份了检查点屏障在输入流中的位置（备份操作发生在检查点屏障被输入算子处理的时候）。**map** 算子接着开始处理检查点屏障，并触发将状态异步备份到稳定存储中这个动作

当 `map` 算子的状态备份和检查点屏障的位置备份被确认之后，该检查点操作就可以被标记为完成，如图 5-6 所示。我们在无须停止或者阻断计算的条件下，在一个逻辑时间点（对应检查点屏障在输入流中的位置）为计算状态拍了快照。通过确保备份的状态和位置指向同一个逻辑时间点，后文将解释如何基于备份恢复计算，从而保证 exactly-once。值得注意的是，当没有出现故障时，Flink 检查点的开销极小，检查点操作的速度由稳定存储的可用带宽决定。回顾数珠子的例子：除了因为数错而需要用到皮筋之外，皮筋会被很快地拨过。（Flink 的开发团队正在研究如何只保存状态的变化，而不保存状态的值，这样做可以让开销变得更小。）

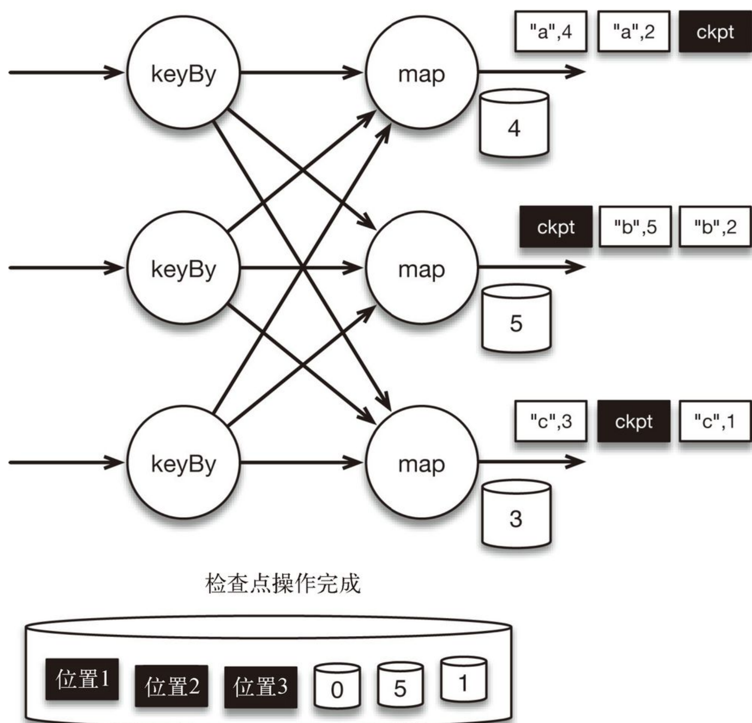


图 **5-6**：检查点操作完成，状态和位置均已备份到稳定存储中。输入流中的所有记录都已处理完成。值得注意的是，备份的状态值与实际的状态值是不同的。备份反映的是检查点的状态

如果检查点操作失败，Flink 会丢弃该检查点并继续正常执行，因为之后的某一个检查点可能会成功。虽然恢复时间可能更长，但是对于状态的保证依旧很有力。只有在一系列连续的检查点操作失败之后，Flink 才会抛出错误，因为这通常预示着发生了严重且持久的错误。

现在来看看图 5-7 所示的情况：检查点操作已经完成，但故障紧随其后。

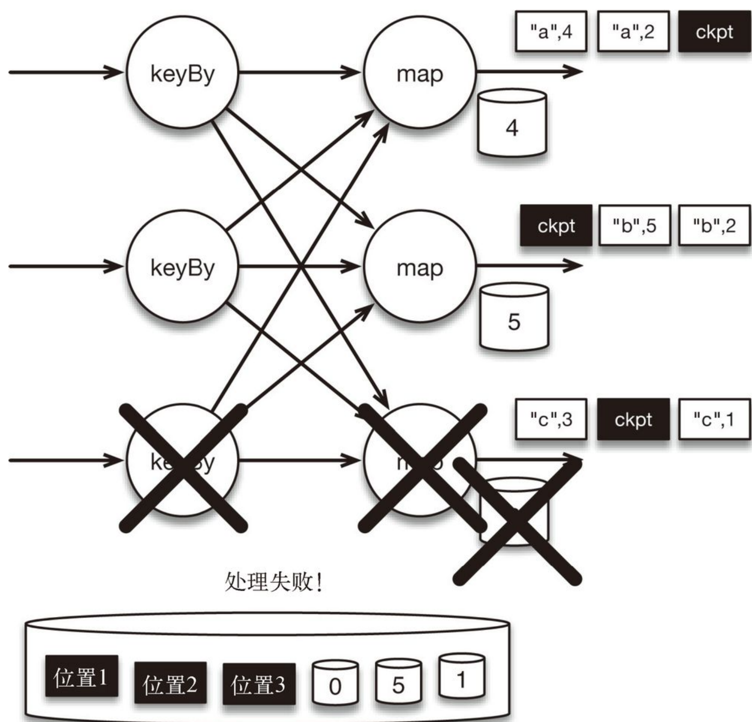


图 **5-7**：故障紧跟检查点，导致最底部的实例丢失

在这种情况下，Flink 会重新拓扑（可能会获取新的执行资源），将输入流倒回到上一个检查点，然后恢复状态值并从该处开始继续计算。在本例中，["a",2]、["a",2] 和 ["c",2] 这几条记录将被重播。

图 5-8 展示了这一重新处理过程。从上一个检查点开始重新计算，可以保证在剩下的记录被处理之后，得到的 `map` 算子的状态值与没有发生故障时的状态值一致。值得注意的是，输出流会含有重复的数据。具体来说，["a",2]、["a",4] 和 ["c",3] 会出现两次。如果 Flink 将输出流写入特殊的输出系统（比如文件系统或者数据库），那么就可以避免这个问题，本章稍后将进一步讨论。

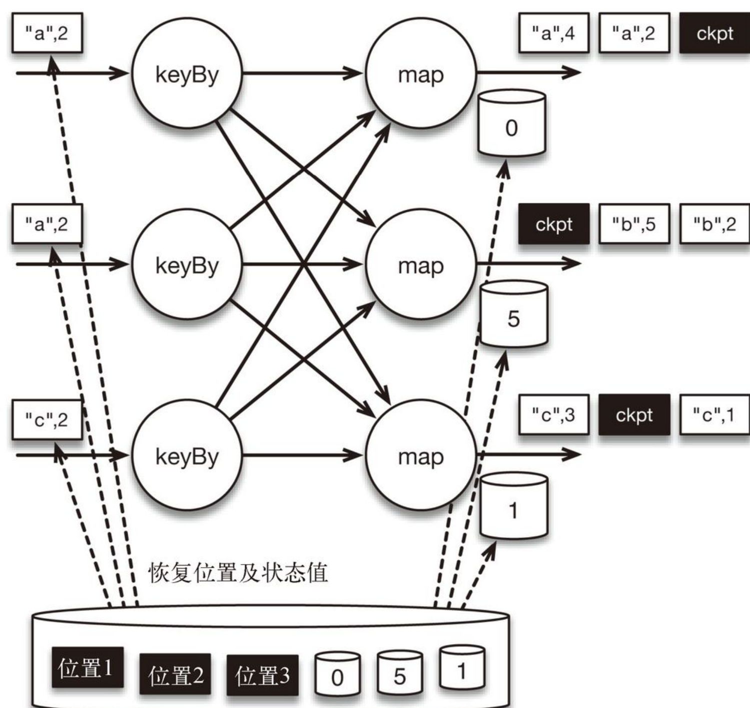


图 **5-8**：**Flink** 将输入流倒回到上一个检查点屏障的位置，同时恢复 **map** 算子的状态值。然后，**Flink** 从此处开始重新处理。这样做保证了在记录被处理之后，**map** 算子的状态值与没有发生故障时的一致

Flink 检查点算法的正式名称是异步屏障快照（asynchronous barrier snapshotting）。该算法大致基于 Chandy-Lamport 分布式快照算法。

5.3　保存点：状态版本控制

检查点由 Flink 自动生成，用来在故障发生时重新处理记录，从而修正状态。Flink 用户还可以通过另一个特性有意识地管理状态版本，这个特性叫作保存点（savepoint）。

保存点与检查点的工作方式完全相同，只不过它由用户通过 Flink 命令行工具或者 Web 控制台手动触发，而不由 Flink 自动触发。和检查点一样，保存点也被保存在稳定存储中。用户可以从保存点重启作业，而不用从头开始。保存点可以被视为作业在某一个特定时间点的快照（该时间点即为保存点被触发的时间点）。

对保存点的另一种理解是，它在明确的时间点保存应用程序状态的版本。这和用版本控制系统保存应用程序的版本很相似。最简单的例子是在不修改应用程序代码的情况下，每隔固定的时间拍快照，即照原样保存应用程序状态的版本，如图 5-9 所示。

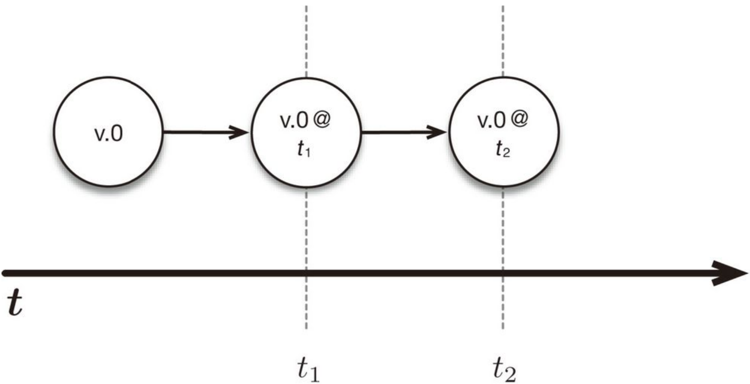


图 **5-9**：手动触发的保存点（以圆圈表示）在不同时间捕获正在运行的 **Flink** 应用程序的状态

在图中，v.0 是某应用程序的一个正在运行的版本。我们分别在 t_1 时刻和 t_2 时刻触发了保存点。因此，可以在任何时候返回到这两个时间点，并且重启程序。更重要的是，可以从保存点启动被修改过的程序版本。举例来说，可以修改应用程序的代码（假设称新版本为 v.1），然后从 t_1 时刻开始运行改动过的代码。这样一来，v.0 和 v.1 这两个版本同时运行，并在之后的时间里获取各自的保存点，如图 5-10 所示。

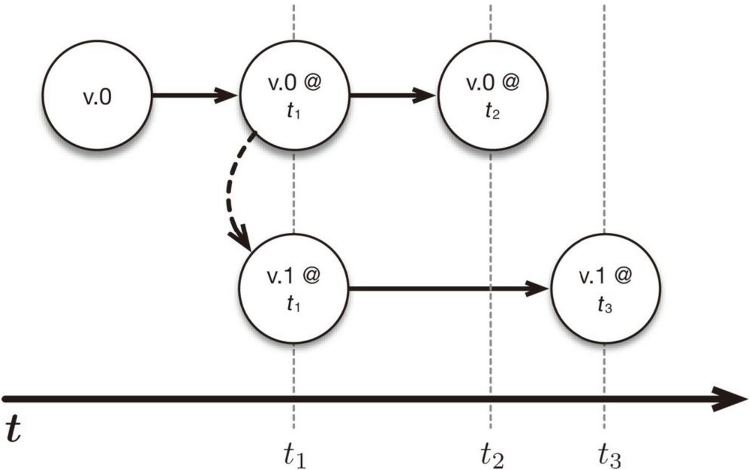


图 **5-10**：使用保存点更新 **Flink** 应用程序的版本。新版本可以从旧版本生成的一个保存点处开始执行

保存点可用于应对流处理作业在生产环境中遇到的许多挑战。

(1) 应用程序代码升级：假设你在已经处于运行状态的应用程序中发现了一个 bug，并且希望之后的事件都可以用修复后的新版本来处理。通过触发保存点并从该保存点处运行新版本，下游的应用程序并不会察觉到不同（当然，被更新的部分除外）。

(2) **Flink** 版本更新：Flink 自身的更新也变得简单，因为可以针对正在运行的任务触发保存点，并从保存点处用新版本的 Flink 重启任务。

(3) 维护和迁移：使用保存点，可以轻松地“暂停和恢复”应用程序。这对于集群维护以及向新集群迁移的作业来说尤其有用。此外，它还有利于开发、测试和调试，因为不需要重播整个事件流。

(4) 假设模拟与恢复：在可控的点上运行其他的应用逻辑，以模拟假设的场景，这样做在很多时候非常有用。

(5) **A/B** 测试：从同一个保存点开始，并行地运行应用程序的两个版本，有助于进行 A/B 测试。

上述所有挑战都真实存在。Flink 内部的检查点机制以保存点的形式呈现给用户，用来应对上述挑战。这反映了 Flink 检查点本质上是一个可持续升级状态版本的可编程机制，这一点很像具有多版本并发控制的数据库系统。下一节在讨论如何提供端到端的一致性时，还会提到检查点机制的这一基本特征。

5.4 端到端的一致性和作为数据库的流处理器

我们已经通过简单的计数例子了解了 Flink 如何保证状态的一致性（即保证 exactly-once）。接下来看看端到端的情况，因为在生产环境中可能会部署这种应用程序（如图 5-11 所示）。

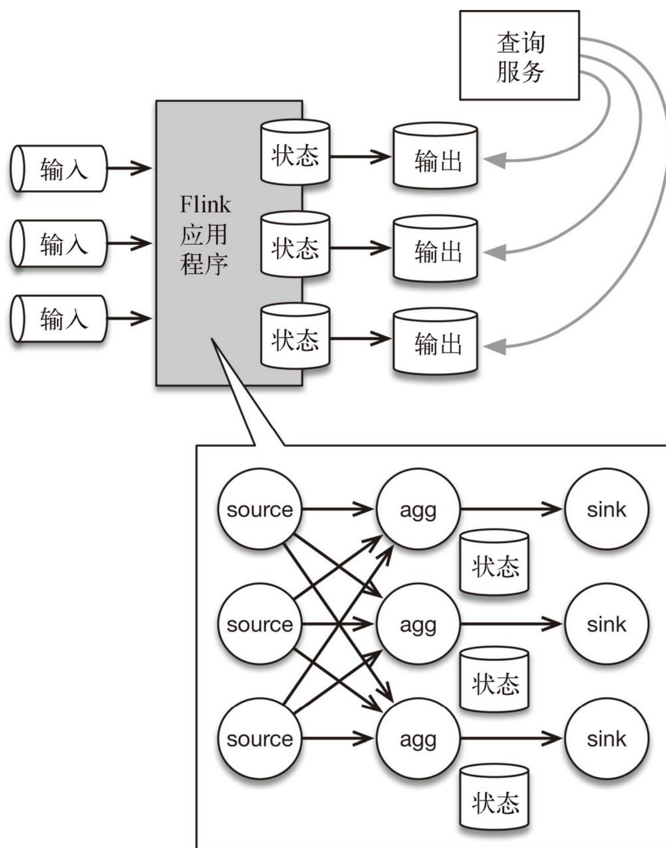


图 **5-11**：在该应用程序架构中，有状态的 **Flink** 应用程序消费来自消息队列的数据，然后将数据写入输出系统，以供查询。底部的详情图展示了 **Flink** 应用程序的内部情况

输入数据来自一个分区存储系统（如 Kafka 或者 MapR Streams 这样的消息队列）。图 5-11 底部的详情图展示了 Flink 拓扑，其中包含 3 个算子。source 读取输入数据，根据 key 分区，并将数据路由到有状态的算子实例（这既可以是 5.2 节提到的 `map` 算子，也可以是窗口聚合算子）。有状态的算子将状态内容（比如前例中的计数结果）或者一些衍生结果写入 sink，再由 sink 将结果传送到输出存储系统中（例如文件系统或数据库）。接着，查询服务（比如数据库查询 API）就可以允许用户对状态进行查询（最简单的例子就是查询计数结果），因为状态已经被写入输出存储系统了。

需要记住的是，在本例中，输出反映的是截至最近一次写入状态之时，Flink 应用程序中的状态内

容。

在将状态内容传送到输出存储系统的过程中，如何保证 exactly-once 呢？这叫作端到端的一致性。本质上有两种实现方法，用哪一种方法则取决于输出存储系统的类型，以及应用程序的需求。

(1) 第一种方法是在 sink 环节缓冲所有输出，并在 sink 收到检查点记录时，将输出“原子提交”到存储系统。这种方法保证输出存储系统中只存在有一致性保障的结果，并且不会出现重复的数据。从本质上说，输出存储系统会参与 Flink 的检查点操作。要做到这一点，输出存储系统需要具备“原子提交”的能力。

(2) 第二种方法是急切地将数据写入输出存储系统，同时牢记这些数据可能是“脏”的，而且需要在发生故障时重新处理。如果发生故障，就需要将输出、输入和 Flink 作业全部回滚，从而将“脏”数据覆盖，并将已经写入输出的“脏”数据删除。注意，在很多情况下，其实并没有发生删除操作。例如，如果新记录只是覆盖旧纪录（而不是添加到输出中），那么“脏”数据只在检查点之间短暂存在，并且最终会被修正过的新数据覆盖。

值得注意的是，这两种方法恰好对应关系数据库系统中的两种为人所熟知的事务隔离级别：已提交读（read committed）和未提交读（read uncommitted）。已提交读保证所有读取（查询输出）都只读取已提交的数据，而不会读取中间、传输中或“脏”的数据。之后的读取可能会返回不同的结果，因为数据可能已被改变。未提交读则允许读取“脏”数据；换句话说，查询总是看到被处理过的最新版本的数据。

某些应用程序可以接受弱一点的语义，所以 Flink 提供了支持多重语义的多种内置输出算子，如支持未提交读语义的分布式文件输出算子。用户可以根据输出存储系统的能力和应用程序的需求选择合适的语义。

根据输出存储系统的类型，Flink 及与之对应的连接器可以一起保证端到端的一致性，并且支持多种隔离级别。

现在回过头看看图 5-11 中的应用程序架构。之所以本例需要有输出存储系统，是因为外部无法访问 Flink 的内部状态，所以输出存储系统成了查询目标。但是，如果可以直接查询状态，则在某些情况下根本就不需要输出存储系统，因为状态本身就已经包含了查询所需的信息。这种情况在许多应用程序中真实存在，直接查询状态可以大大地

简化架构，同时大幅提升性能，如图 5-12 所示。

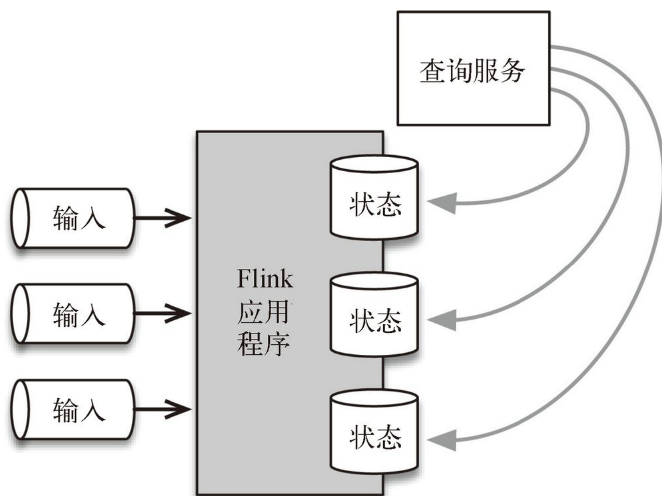


图 **5-12**：利用 **Flink** 的可查询状态特性可以简化应用程序架构。对于状态本身就是所需信息的查询来说，直接查询状态可以提升性能

Flink 社区正致力于完善可查询状态特性。Flink 提供一个查询 API，通过该 API 可以对 Flink 发出查询请求，然后得到当前的状态值。从某种意义上说，在有限的情景下，Flink 可以替代数据库，并同时提供写路径（输入流不断更新状态）和读路径（可查询状态）。尽管这对于许多应用程序都行得通，但可查询状态受到的限制还是比通用数据库大得多。

5.5　Flink的性能

Flink 的性能测试基于 Yahoo! Streaming Benchmark 及其一系列变体进行。

5.5.1　Yahoo! Streaming Benchmark

2015 年 12 月，Yahoo! 的 Storm 团队发表了一篇博客文章 **1**，并在其中展示了 Storm、Flink 和 Spark Streaming 的性能测试结果。该测试对于业界而言极具价值，因为它是流处理领域的第一个基于真实应用程序的基准测试。

1https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-

computation-engines-at

该应用程序从 Kafka 消费广告曝光消息，从 Redis 查找每个广告对应的广告宣传活动，并按照广告宣传活动分组，以 10 秒为窗口计算广告浏览量。10 秒窗口的最终结果被存储在 Redis 中，这些窗口的状态也按照每秒记录一次的频率被写入 Redis，以方便用户对它们进行实时查询。在最初的性能测评中，因为 Storm 是无状态流处理器（即它不能定义和维护状态），所以 Flink 作业也按照无状态模式编写。所有状态都被存储在 Redis 中，如图 5-13 所示。

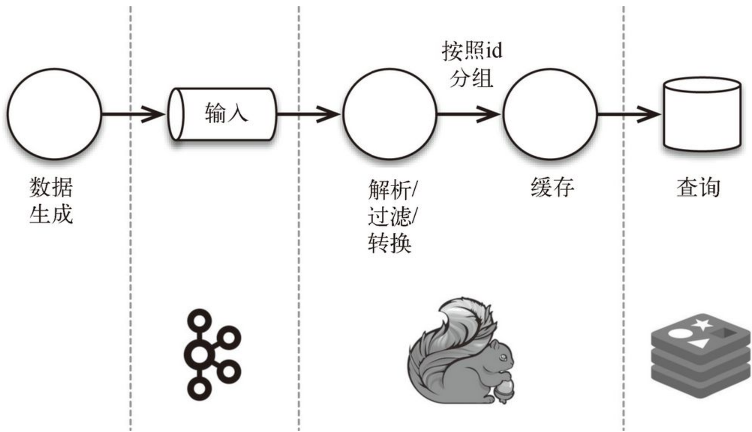


图 **5-13**：**Yahoo! Streaming Benchmark** 所采用的作业。被测试的流处理器有 **Storm**、**Flink** 和 **Spark Streaming**（本图中只有 **Flink** 的 **logo**）

图 5-14 展示了测试结果。

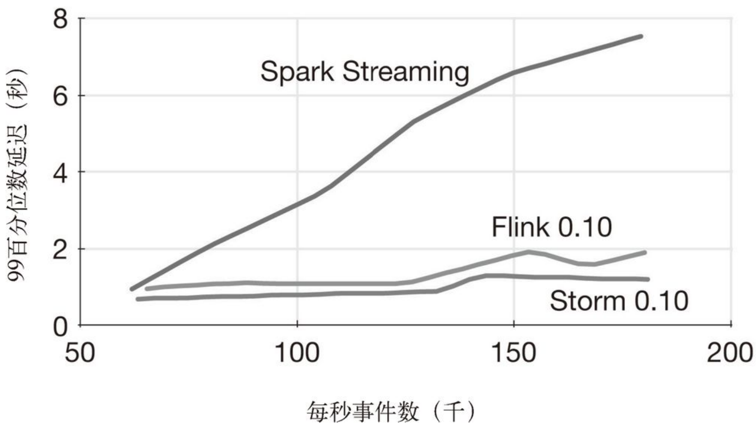


图 **5-14**：**Yahoo! Streaming Benchmark** 的结果。横轴表示每秒的事件吞吐量，以千为单位。纵轴表示端到端的 **99** 百分位数延迟（即 **99%** 的事件在延迟时间段内到达），以秒为单位。详见博客文章 ***Benchmarking Streaming Computation Engines at Yahoo!*** **2**

> 2https://yahooeng.tumblr.com/post/135321837876/benchmarking-streaming-computation-engines-at

如图 5-14 所示，在性能测评中，Spark Streaming 遇到了吞吐量和延迟性难两全的问题。随着批处理作业规模的增加，延迟升高。如果为了降低延迟而缩减规模，吞吐量就会减少。Storm 和 Flink 则可以在吞吐量增加时维持低延迟。

为了进一步测试 Flink 的性能，测试人员设置了一系列不同的场景，并逐步测试。

5.5.2　变化1：使用Flink状态

最初的性能测评专注于在相对较低的吞吐量下，测量端到端的延迟，即使在极限状态下，也不关注容错性。此外，应用程序中的 key 基数非常小（100），这使得测试结果无法反映用户量大的情况，或者 key 空间随着时间增长的情况（如 tweet）。2016 年 2 月，data Artisans 的博客发表了一篇文章 3，对 Yahoo! Streaming Benchmark 进行了拓展，并专注于解决上述问题。由于最初的测试结果显示 Spark Streaming 的性能欠佳，因此这次的测试对象只有 Storm 和 Flink，它们在最初的测试中有着类似的表现。

> 3https://data-artisans.com/blog/extending-the-yahoo-streaming-benchmark

第 1 个变化是利用 Flink 提供的状态容错特性重新实现应用程序，如图 5-15 所示。这使得应用程序能保证 exactly-once。

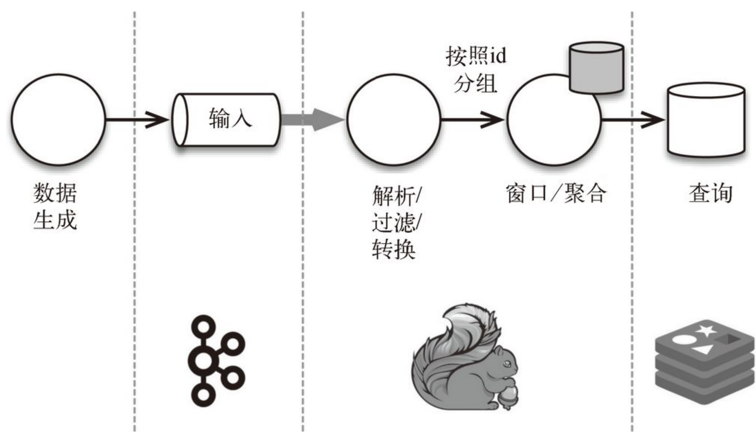


图 **5-15**：重新实现的应用程序利用了 **Flink** 内置的状态机制，并且可以保持每秒 **300** 万事件的吞吐量，同时保证 **exactly-once**。此时，应用程序的瓶颈在于 **Flink** 集群与 **Kafka** 集群的连接（图中以粗箭头表示）

5.5.3　变化2：改进数据生成器并增加吞吐量

第 2 个变化是通过用每秒可以生成数百万事件的数据生成器来增加输入流的数据量。结果如图 5-16 所示。

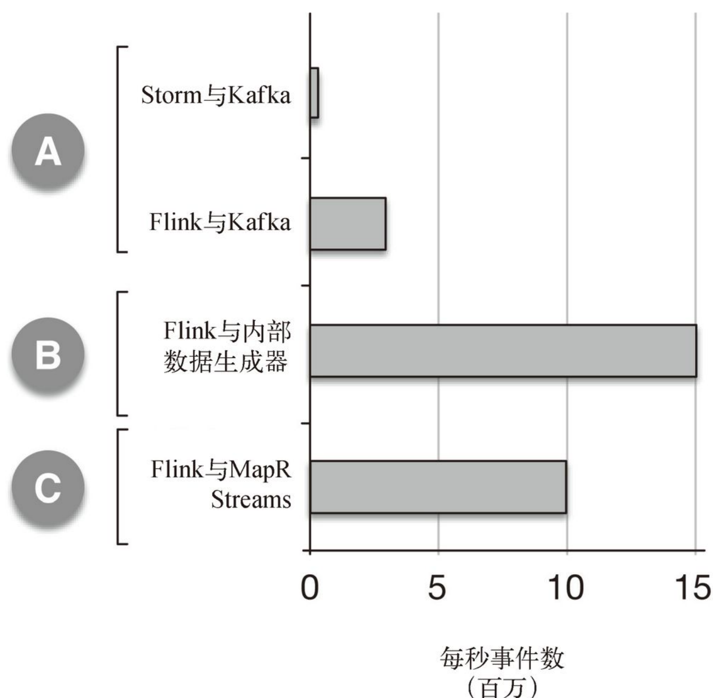


图 **5-16**：使用高吞吐数据生成器的结果：（**A**）当 **Storm** 与 **Kafka** 一起使用时，应用程序可以保持每秒 **40** 万事件的处理速度，并且瓶颈在于 **CPU**；当 **Flink** 与 **Kafka** 一起使用时，应用程序可以保持每秒 **300** 万事件的处理速度，并且瓶颈在于网络；（**B**）当消除网络瓶颈时，**Flink** 应用程序可以保持每秒 **1500** 万事件的处理速度；（**C**）在额外的测试中，消息队列由 **MapR Streams** 提供，并且采用 **10** 个高性能网络节点（硬件与前两种情况中的不同）；**Flink** 应用程序可以保持每秒 **1000** 万事件的处理速度

Storm 能够承受每秒 40 万事件，但受限于 CPU；Flink 则可以达到每秒 300 万事件（7.5 倍），但受限于 Kafka 集群和 Flink 集群之间的网络。

5.5.4　变化3：消除网络瓶颈

为了看看在没有网络瓶颈问题时 Flink 的性能如何，我们将数据生成器移到 Flink 应用程序的内部。图 5-17 展示了这个流程。在这样的条件下，Flink 可以保持每秒 1500 万事件的处理速度（这是 Storm 的 37.5 倍），如图 5-16 所示。将数据生成器整合到 Flink 应用程序中，可以测试性能极限，但这种做法并不现实，因为现实世界中的数

据必须从应用程序的外部流入。

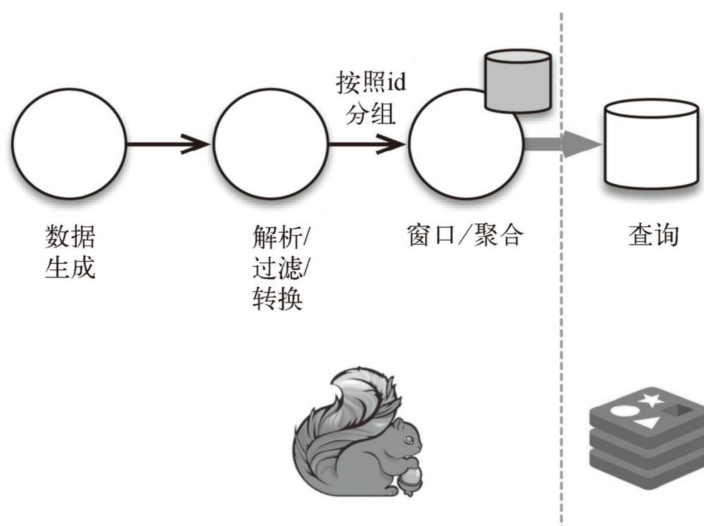


图 **5-17**：通过将数据生成器整合到 **Flink** 应用程序中，可以消除网络瓶颈，并且使系统支撑每秒 **1500** 万事件的吞吐量。在增加 **key** 基数之后，瓶颈又转移到每秒对 **Redis** 的写入上。该测试并不符合生产环境的配置，它的目的是测试 **Flink** 的极限

值得注意的是，这绝对不是 Kafka 的极限（Kafka 可以支撑比这更大的吞吐量），而仅仅是测试所用的硬件环境的极限——Kafka 集群和 Flink 集群之间的网络连接太慢。

5.5.5　变化4：使用MapR Streams

另一种避免网络瓶颈并测试 Flink 性能的方法是使用 MapR Streams。在另一个测试中，同样的 Flink 应用程序通过 MapR Streams 接收数据。

使用 MapR Streams 之后，流处理被整合进整个平台，从而使得 Flink 可以与数据生成任务和数据传输任务一起运行，这样就避免了连接 Kafka 集群和 Flink 集群时遇到的大部分问题。在这种高性能配置和更快的网络硬件环境下，Flink 能够支撑每秒 1000 万事件的处理速度。

5.5.6　变化5：增加key基数

最后一个变化是增加 key 基数（广告宣传活动的数量）。在最初的测试中，key 基数只有 100。这些 key 每秒都会被写入 Redis，以供查询。当 key 基数增加到 100 万时，系统的整体吞吐量减少到每秒 28 万事件，因为向 Redis 写入成了系统瓶颈。使用 Flink 可查询状态的一个早期原型（如图 5-18 所示），可以消除这种瓶颈，使系统的处理速度恢复到每秒 1500 万事件，并且有 100 万个 key 可供查询，如图 5-19 所示。

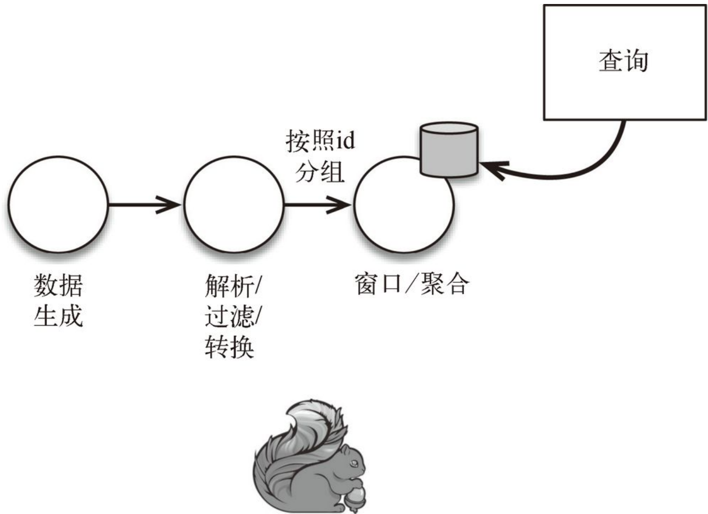


图 **5-18**：为 **key** 基数较大的场景消除系统瓶颈

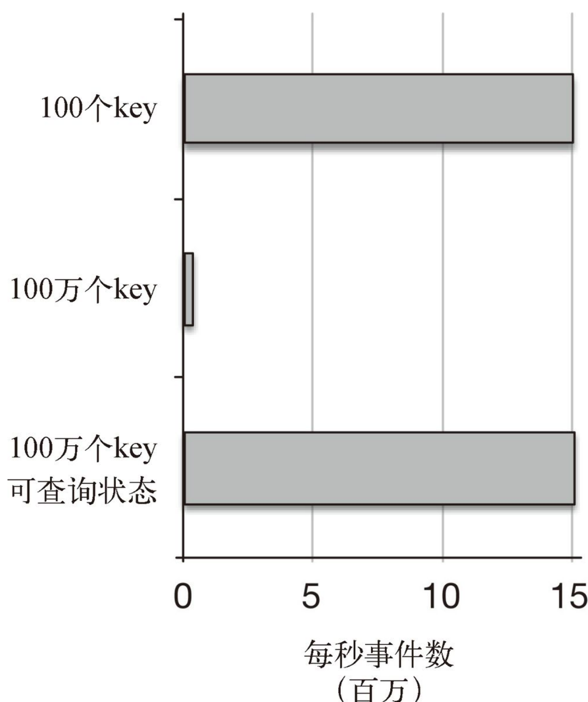


图 **5-19**：通过将查询功能移入 **Flink** 可查询状态的一个原型，系统甚至可以在 **key** 基数非常大的情况下仍然维持每秒 **1500** 万事件的处理速度

本例说明了什么呢？通过避免流处理瓶颈，同时利用 Flink 的有状态流处理能力，可以使吞吐量达到 Storm 的 30 倍左右，同时还能保证 exactly-once 和高可用性。大致来说，这意味着与 Storm 相比，Flink 的硬件成本或云计算成本仅为前者的 1/30，同样的硬件能处理的数据量则是前者的 30 倍。

5.6　结论

在本章中，我们看到有状态的流处理如何改变了游戏规则。通过 Flink 的检查点机制，能够同时实现容错、高吞吐和低延迟。这完全避免了人们曾经认为无法避免的权衡，并体现了 Flink 最重要的一个优势。

Flink 的另一个优势是，它用同一种技术实现流处理和批处理，因此完全不必再建一个批处理层。第 6 章将简述 Flink 如何实现批处理。

第 6 章　批处理：一种特殊的流处理

到目前为止，本书讨论的都是无限流处理，即从某个时间点开始，持续不停地处理数据，如图 6-1 所示。

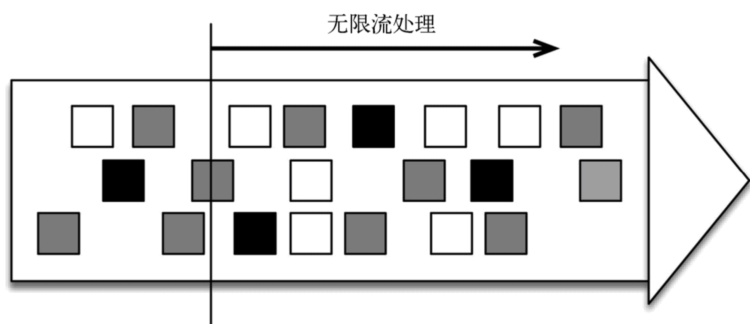


图 **6-1**：无限流处理：输入数据没有尽头；数据处理从当前或者过去的某一个时间点开始，持续不停地进行

另一种处理形式叫作有限流处理，即从某一个时间点开始处理数据，然后在另一个时间点结束，如图 6-2 所示。输入数据可能本身是有限的（即输入数据集并不会随着时间增长），也可能出于分析的目的被人为地设定为有限集（即只分析某一个时间段内的事件）。

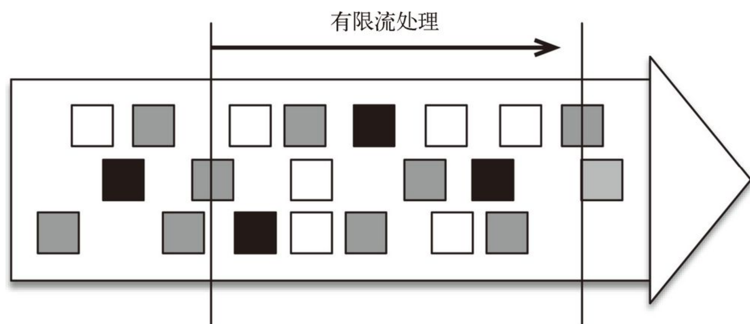


图 **6-2**：有限流处理：输入数据有头有尾；数据处理在一段时间后停止

显然，有限流处理是无限流处理的一种特殊情况，它只不过在某个时间点停止而已。此外，如果计算结果不在执行过程中连续生成，而仅在末尾处生成一次，那就是批处理（分批处理数据）。

批处理是流处理的一种非常特殊的情况。在流处理中，我们为数据定义滑动窗口或滚动窗口，并且在每次窗口滑动或滚动时生成结果。批处理则不同，我们定义一个全局窗口，所有的记录都属于同一个窗口。举例来说，以下代码表示一个简单的 Flink 程序，它负责每小时对某网站的访问者计数，并按照地区分组。

```
val counts = visits
  .keyBy("region")
  .timeWindow(Time.hours(1))
  .sum("visits")
```

如果知道输入数据是有限的，则可以通过以下代码实现批处理。

```
val counts = visits
  .keyBy("region")
  .window(GlobalWindows.create)
  .trigger(EndOfTimeTrigger.create)
  .sum("visits")
```

Flink 的不寻常之处在于，它既可以将数据当作无限流来处理，也可以将它当作有限流来处理。Flink 的 DataSet API 就是专为批处理而生的，如下所示。

```
val counts = visits
  .groupBy("region")
  .sum("visits")
```

如果输入数据是有限的，那么以上代码的运行结果将与前一段代码的相同，但是它对于习惯使用批处理器的程序员来说更友好。

6.1　批处理技术

从原则上说，批处理是一种特殊的流处理：当输入数据是有限的，并且只需要得到最终结果时，对所有数据定义一个全局窗口并在窗口里进行计算即可。但是，这样做的效率如何呢？

传统上，有限数据流由专用的批处理器处理；某些时候，它比流处理器更高效。但是，在流处理器中整合高效、大规模的批处理所需的大部分优化方案，是可行的做法。这正是 Flink 所做的工作，而且这样做的效率很高。

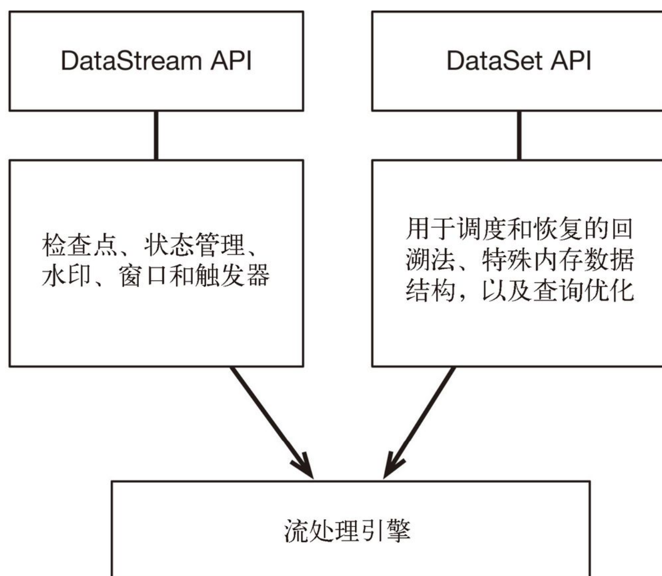


图 **6-3**：**Flink** 通过一个底层引擎同时支持流处理和批处理

同样的后端（流处理引擎）被用来处理有限数据和无限数据。在流处理引擎之上，Flink 有以下机制：

- 检查点机制和状态机制：用于实现容错、有状态的处理；
- 水印机制：用于实现事件时钟；
- 窗口和触发器：用于限制计算范围，并定义呈现结果的时间。

在同一个流处理引擎之上，Flink 还存在另一套机制，用于实现高效的批处理。尽管本书并不打算详细讨论这些机制，但是仍要指出以下重点：

- 用于调度和恢复的回溯法：由 Microsoft Dryad 引入，现在几乎用于所有批处理器；
- 用于散列和排序的特殊内存数据结构：可以在需要时，将一部分数据从内存溢出到硬盘上；
- 优化器：尽可能地缩短生成结果的时间。

在写作本书时，以上两套机制分别对应各自的 API（DataStream API 和 DataSet API）；在创建 Flink 作业时，并不能通过将两者混合在一起来同时利用 Flink 的所有功能。然而，这并不是问题。实际上，Flink 社区已经在研究如何统一这两个 API。Apache Beam 社区已经创建出了同时适用于流处理和批处理的 API，它可以生成用于执行的 Flink 程序。

6.2　案例研究：Flink作为批处理器

在 2015 年的 Flink Forward 研讨会上，Dongwon Kim1 展示了他所做的性能测试。他对 MapReduce、Tez、Spark 和 Flink 在执行纯批处理任务时的性能做了比较。测试的批处理任务是 TeraSort 和分布式散列连接。

> 1他当时是韩国浦项科技大学的博士后研究员。

第一个任务是 TeraSort，即测量为 1TB 数据排序所用的时间。就上述系统而言，TeraSort 本质上是分布式排序问题，如图 6-4 所示。它由以下几个阶段组成：

(1) 读取阶段：从 HDFS 文件中读取数据分区；

(2) 本地排序阶段：对上述分区进行部分排序；

(3) 混洗阶段：将数据按照 key 重新分布到处理节点上；

(4) 终排序阶段：生成排序输出；

(5) 写入阶段：将排序后的分区写入 HDFS 文件。

典型的分布式排序阶段

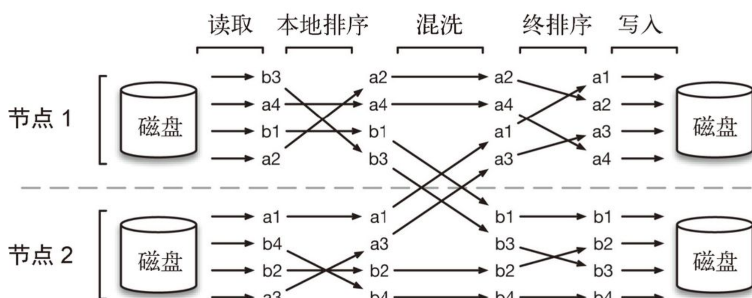


图 **6-4**：分布式排序的处理阶段

Hadoop 发行版包含对 TeraSort 的实现，同样的实现也可以用于 Tez，因为 Tez 可以执行通过 MapReduce API 编写的程序。Spark 和 Flink 的 TeraSort 实现由 Dongwon Kim 提供 **2**。用来测量的集群由 42 台机器组成，每台机器包含 12 个 CPU 内核、24GB 内存，以及 6 块硬盘。

> **2**https://github.com/eastcirclek/terasort

图 6-5 展示了测试结果。结果显示，Flink 的排序时间比其他所有系统都少。MapReduce 用了 2157 秒，Tez 用了 1887 秒，Spark 用了 2171 秒，Flink 则只用了 1480 秒。

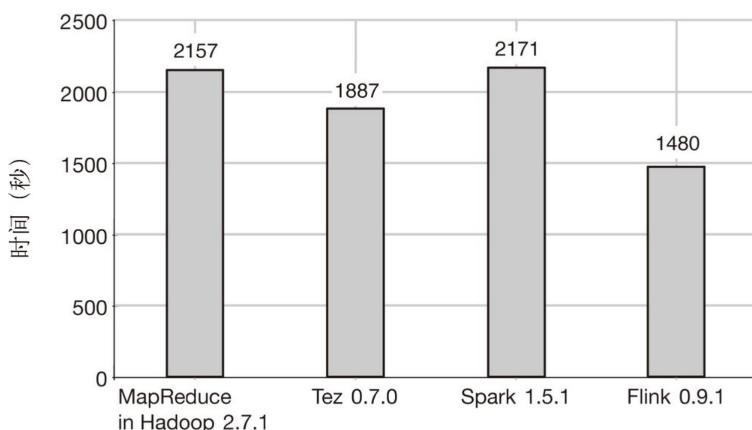


图 **6-5**：**MapReduce**、**Tez**、**Spark** 和 **Flink** 分别对应的 **TeraSort** 结果

第二个任务是一个大数据集（240GB）和一个小数据集（256MB）之间的分布式散列连接。结果显示，Flink 仍然是速度最快的系统，它所用的时间分别是 Tez 和 Spark 的 1/2 和 1/4，如图 6-6 所示。

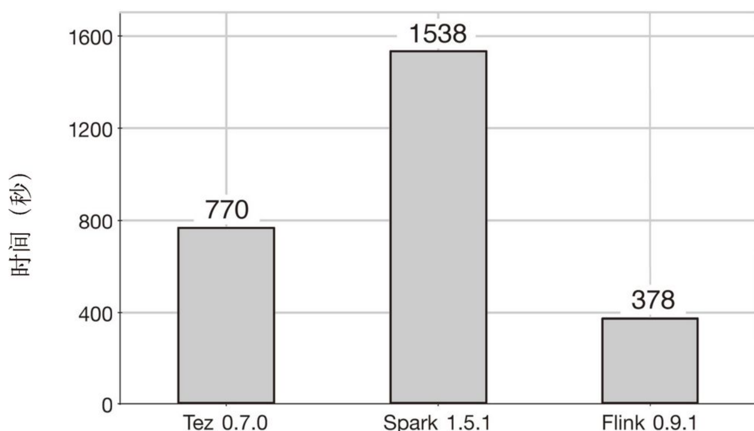


图 6-6：Tez、Spark 和 Flink 分别对应的散列连接结果

产生以上结果的总体原因是，Flink 的执行过程是基于流的，这意味着各个处理阶段有更多的重叠，并且混洗操作是流水线式的，因此磁盘访问操作更少。相反，MapReduce、Tez 和 Spark 是基于批的，这意味着数据在通过网络传输之前必须先被写入磁盘。该测试说明，在使用 Flink 时，系统空闲时间和磁盘访问操作更少。

值得一提的是，性能测试结果中的原始数值可能会因集群设置、配置和软件版本而异。如果现在再测试一遍，那么得到的数值的确可能不一样（上述测试用到的软件版本分别是：Hadoop 2.7.1、Tez 0.7.0、Spark 1.5.1，以及 Flink 0.9.1，这些系统在如今都有了更新的版本）。本节的重点是，即使是批处理器所擅长的任务，流处理器（Flink）在经过适当的优化后也仍然可以表现得和批处理器（MapReduce、Tez 和 Spark）一样好，甚至更好。因此，Flink 可以用同一个数据处理框架来处理无限数据流和有限数据流，并且不会牺牲性能。

附录　其他资源

进一步使用Flink

我们希望你已经跃跃欲试，并且准备好使用 Flink 了。从哪里开始呢？答案是 Flink 的网站：https://flink.apache.org。该网站有“快速入门”指南，通过例子教你如何使用 Flink 摄取和分析维基百科的编辑日志。只需花几分钟，你就可以开始编写你的第一个流处理程序了。

如果你偏爱视觉效果，可以看看 MapR 公司提供的例子：如何用 Flink 摄取纽约市出租车路线的数据流，并用 Kibana 将它可视化，详见 *The Essential Guide to Streaming-first Processing with Apache Flink*（https://www.mapr.com/blog/essential-guide-streaming-first-processing-apache-flink）。

若想进行更深入的学习，可以访问 data Artisans 免费提供的培训资源：http://training.data-artisans.com。其上的所有幻灯片、练习和解决方案都是开源的。

关于时间和窗口的更多内容

本书用很大的篇幅讨论了时间和窗口的多个方面，并解释了 Flink 的工作原理以及在使用时可以做的选择。这些主题也是一系列博客文章所讨论的内容。如果你想更深入地了解 Flink 窗口的工作原理，可以参考 http://flink.apache.org/news/2015/12/04/Introducing-windows.html；若想了解关于会话窗口的更多细节，可以参考 http://data-artisans.com/blog/session-windowing-in-flink/；若想进一步了解 Flink 窗口和水印机制，以及事件时间适用于哪些应用程序，请参考 http://data-artisans.com/blog/how-apache-flink-enables-new-streaming-applications-part-1。

关于状态和检查点的更多内容

若想深入了解 Flink 的检查点机制，以及它如何实现容错性流处理，请参考 http://data-artisans.com/blog/high-throughput-low-latency-and-exactly-once-stream-processing-with-apache-flink。

若想了解关于保存点的更多信息，请观看这个短视频：https://mapr.com/blog/savepoints-apache-flink-stream-processing-whiteboard-walkthrough/。在该视频中，data Artisans 的首席技术官 Stephan Ewen 解释了如何用保存点重播流数据。在重新处理数

据、修复 bug 和进行更新时，保存点很有用。

更多关于保存点的内容，请参考 http://data-artisans.com/blog/how-apache-flink-enables-new-streaming-applications/。

若想了解基于 Yahoo! Streaming Benchmark 所做的拓展，请访问 https://data-artisans.com/blog/extending-the-yahoo-streaming-benchmark/。

用Flink进行批处理

若想了解流处理器如何进行批处理，请访问 http://data-artisans.com/blog/batch-is-a-special-case-of-streaming。

Flink 博客上也有许多信息。若想深入了解 Flink 为了优化批处理而使用的具体机制，请访问以下网址。

- http://flink.apache.org/news/2015/05/11/Juggling-with-Bits-and-Bytes.html
- http://flink.apache.org/news/2015/03/13/peeking-into-Apache-Flinks-Engine-Room.html
- http://data-artisans.com/blog/computing-recommendations-at-extreme-scale-with-apache-flink/

Flink用例和用户故事

经常使用 Flink 的公司会发表一些文章，并在其中描述收获和用法。以下是一些推荐的用户故事。

- https://techblog.king.com/rbea-scalable-real-time-analytics-king/
- https://tech.zalando.de/blog/apache-showdown-flink-vs.-spark/
- http://data-artisans.com/blog/flink-at-bouygues-html/
- http://data-artisans.com/blog/how-we-selected-apache-flink-at-otto-group/

人们每年在 Flink Forward 研讨会上演示的视频和幻灯片，都是了解各个公司如何使用 Flink 的绝佳资源，详见 Flink Forward 的网站。

流处理架构

若想深入了解流处理架构，以及 Kafka 和 MapR Streams 所用的消息传输技术，推荐阅读 Ted Dunning 和 Ellen Friedman 合著的书 *Streaming Architecture*。

以下两个短视频解释了流处理架构在支持微服务方面的优势。

- https://www.mapr.com/blog/key-requirements-streaming-platforms-micro-services-advantage-whiteboard-walkthrough-part-1
- https://www.mapr.com/blog/streaming-data-how-move-state-flow-whiteboard-walkthrough-part-2

消息传输：Kafka

若想尝试使用 Kafka，可以在 MapR 公司网站上的这篇博客文章里找到示例程序：https://www.mapr.com/blog/getting-started-sample-programs-apache-kafka-09。

此外，推荐阅读由 Neha Narkhede、Gwen Shapira 和 Todd Palino 合著的书《Kafka 权威指南》[1]。

> 1详见 http://www.ituring.com.cn/book/2067。——编者注

消息传输：MapR Streams

MapR Streams 是 MapR 融合数据平台的一个主要部分。要了解关于它的更多内容，请参考下列资源。

- MapR Streams 的功能概览，包括流层面上的管理和跨地域的流复制能力：https://www.mapr.com/products/mapr-streams。
- MapR Streams 的示例程序（用 Kafka API）：https://www.mapr.com/blog/getting-started-sample-programs-mapr-streams。
- 大致比较 Kafka 和 MapR Streams：https://mapr.com/blog/apache-kafka-and-mapr-streams-terms-techniques-and-new-designs。

Ted Dunning和Ellen Friedman的部分著作

- *Streaming Architecture: New Designs Using Apache Kafka and MapR Streams*
- *Sharing Big Data Safely: Managing Data Security*
- *Real-World Hadoop*
- *Time Series Databases: New Ways to Store and Access Data*
- *Practical Machine Learning: A New Look at Anomaly Detection*
- *Practical Machine Learning: Innovations in Recommendation*

关于作者

埃伦 • 弗里德曼（Ellen Friedman）既是解决方案咨询师，也是著名的演讲者和作者。她目前的写作重点是大数据。此外，她还是 Apache Drill 和 Apache Mahout 这两个项目的贡献者。她拥有生物化学博士学位，具有多年的科学研究经验，著作涵盖许多技术主题，包括分子生物学、非传统性遗传方式，以及海洋学。埃伦还是魔法主题卡通书 *A Rabbit Under the Hat* 的合著者。她的 Twitter 用户名是 @Ellen_Friedman。

科斯塔斯 • 宙马斯（Kostas Tzoumas）是 data Artisans 公司的联合创始人兼首席执行官。data Artisans 公司由 Apache Flink 的创始团队创办。科斯塔斯是 Apache Flink 项目管理委员会的成员，拥有丹麦奥尔堡大学的计算机科学博士学位。他是多篇技术论文和博客文章的作者，写作主题是流处理和数据科学。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_interview，讲述码农精彩人生
- 微信 图灵教育：turingbooks

091507240605ToBeReplacedWithUserId

版权信息

书名：数据科学实战

作者：Rachel Schutt , Cath O'Neil

译者：冯凌秉 王群锋

ISBN：978-1-449-35865-5

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

091507240605ToBeReplacedWithUserId

版权声明
O'Reilly Media, Inc.介绍
    业界评论
作者介绍
关于封面图
前言
    初衷
    课程的起源
    本书的起源
    本书内容
    组织结构
    阅读须知
    书中的代码
    目标读者
    基础知识要求
    补充阅读
    关于本书其他贡献者
    排版约定
    使用代码示例
    Safari® Books Online
    联系我们
    致谢
第 1 章　简介：什么是数据科学
    1.1　大数据和数据科学的喧嚣
    1.2　冲出迷雾
    1.3　为什么是现在
        数据化
    1.4　数据科学的现状和历史
        数据科学的职位
    1.5　数据科学的知识结构
    1.6　思维实验：元定义
    1.7　什么是数据科学家

1.7.1　学术界对数据科学家的定义
1.7.2　工业界对数据科学家的定义
第 2 章　统计推断、探索性数据分析和数据科学工作流程
2.1　大数据时代的统计学思考
2.1.1　统计推断
2.1.2　总体和样本
2.1.3　大数据的总体和样本
2.1.4　大数据意味着大胆的假设
2.1.5　建模
2.2　探索性数据分析
2.2.1　探索性数据分析的哲学
2.2.2　练习：探索性数据分析
2.3　数据科学的工作流程
数据科学家在数据科学工作流程中的角色
2.4　思维实验：如何模拟混沌
2.5　案例学习：RealDirect
2.5.1　RealDirect是如何赚钱的
2.5.2　练一练：RealDirect公司的数据策略
第 3 章　算法
3.1　机器学习算法
3.2　三大基本算法
3.2.1　线性回归模型
3.2.2　k近邻模型（k-NN）
3.2.3　k均值算法
3.3　练习：机器学习算法基础
答案
3.4　总结
3.5 思维实验：关于统计学家的自动化
第 4 章　垃圾邮件过滤器、朴素贝叶斯与数据清理
4.1　思维实验：从实例中学习
4.1.1　线性回归为何不适用
4.1.2　k近邻效果如何

4.2　朴素贝叶斯模型
4.2.1　贝叶斯法则
4.2.2　个别单词的过滤器
4.2.3　直通朴素贝叶斯
4.3　拉普拉斯平滑法
4.4　对比朴素贝叶斯和k 近邻
4.5　Bash代码示例
4.6　网页抓取：API和其他工具
4.7　Jake的练习题：文章分类问题中的朴素贝叶斯模型
使用《纽约时报》的API： R代码示例
第 5 章　逻辑回归
5.1　思维实验
5.2　分类器
5.2.1　运行时间
5.2.2　你自己
5.2.3　模型的可解释性
5.2.4　可扩展性
5.3　逻辑回归：一个来自M6D的真实案例研究
5.3.1　点击模型
5.3.2　模型背后
5.3.3　和的参数估计
5.3.4　牛顿法
5.3.5　随机梯度下降法
5.3.6　操练
5.3.7　模型评价
5.4　练习题
示例R代码
第 6 章　时间戳数据与金融建模
6.1　Kyle Teague与GetGlue公司
6.2　时间戳
6.2.1　探索性数据分析（EDA）
6.2.2　指标和新变量

6.2.3　下一步怎么做
6.3　轮到Cathy O'Neill了
6.4　思维实验
6.5　金融建模
6.5.1　样本期内外以及因果关系
6.5.2　金融数据处理
6.5.3　对数收益率
6.5.4　实例：标准普尔指数
6.5.5　如何衡量波动率
6.5.6　指数平滑法
6.5.7　金融模型的反馈
6.5.8　聊聊回归模型
6.5.9　先验信息量
6.5.10　一个小例子
6.6　练习：GetGlue提供的时间戳数据
练习：金融建模
第 7 章　从数据到结论
7.1　William Cukierski
7.1.1　背景介绍：数据科学竞赛
7.1.2　背景介绍：众包模式
7.2　Kaggle模式
7.2.1　Kaggle的参赛者
7.2.2　Kaggle的客户
7.3　思维实验：关于作业自动评分系统
7.4　特征选择
7.4.1　例子：留住用户
7.4.2　过滤型
7.4.3　包装型
7.4.4　决策树与嵌入型变量选择
7.4.5　熵
7.4.6　决策树算法
7.4.7　如何在决策树模型中处理连续性变量

7.4.8　随机森林
7.4.9　用户黏性：模型的预测能力与可解释性
7.5　David Huffaker：谷歌社会学研究的新方法
7.5.1　从描述性统计到预测模型
7.5.2　谷歌的社交研究
7.5.3　隐私保护
7.5.4　思维实验：如何消除用户的顾虑
第 8 章　构建面向大量用户的推荐引擎
8.1　一个真实的推荐引擎
8.1.1　最近邻算法回顾
8.1.2　最近邻模型的已知问题
8.1.3　超越近邻模型：基于机器学习的分类模型
8.1.4　高维度问题
8.1.5　奇异值分解（SVD）
8.1.6　关于SVD的重要特性
8.1.7　主成分分析（PCA）
8.1.8　交替最小二乘法
8.1.9　固定矩阵V，更新矩阵U
8.1.10　关于这些算法的一点思考
8.2　思维实验：如何过滤模型中的泡沫
8.3　练习：搭建自己的推荐系统
Python示例代码
第 9 章　数据可视化与欺诈侦测
9.1　数据可视化的历史
9.1.1　Gabriel Tarde
9.1.2　Mark的思维实验
9.2　到底什么是数据科学
9.2.1　Processing
9.2.2　Franco Moretti
9.3　一个数据可视化的方案实例
9.4　Mark的数据可视化项目
9.4.1　《纽约时报》大厅里的可视化：Moveable Type

9.4.2　屏幕上的生命：Cascade可视化项目
9.4.3　Cronkite广场项目
9.4.4　eBay与图书网购
9.4.5　公共剧场里的“莎士比亚机”
9.4.6　这些展览的目的是什么
9.5　数据科学和风险
9.5.1　关于Square公司
9.5.2　支付风险
9.5.3　模型效果的评估问题
9.5.4　建模小贴士
9.6　数据可视化在Square
9.7　Ian的思维实验
9.8　关于数据可视化
数据可视化练习作业
第 10 章　社交网络与数据新闻学
10.1　Morning Analytics与社交网络
案例-属性数据与社交网络数据
10.2　社交网络分析
10.3　关于社交网络分析的相关术语
10.3.1　如何衡量向心性
10.3.2　使用哪种向心性测度
10.4　思维实验
10.5　Morningside Analytics
可视化与中观视角
10.6　从统计学的角度看社交网络分析
10.6.1　网络的表示方法与特征值向心度
10.6.2　随机网络的第一个例子：Erdos-Renyi模型
10.6.3　随机网络的第二个例子：指数随机网络图模型
10.7　数据新闻学
10.7.1　关于数据新闻学的历史回顾
10.7.2　数据新闻报告的写作：来自专家的建议
第 11 章　因果关系研究

11.1　相关性并不代表因果关系
11.1.1　对因果关系提问
11.1.2　干扰因子：一个关于在线约会网站的例子
11.2　OK Cupid的发现
11.3　黄金准则：随机化临床实验
11.4　A/B 测试
11.5　退一步求其次：关于观察性研究
11.5.1　辛普森悖论
11.5.2　鲁宾因果关系模型
11.5.3　因果关系的可视化
11.5.4　定义：因果关系
11.6　三个小建议
第 12 章　流行病学
12.1　Madigan的学术背景
12.2　思维实验
12.3　统计学在现代
12.4　医学文献与观察性研究
12.5　分层法不解决干扰因子的问题
人们在实证中到底如何处理干扰因子的问题
12.6　就没有更好的办法吗
12.7　研究性实验（OMOP）
12.8　最后的思维实验
第 13 章　从竞赛中学到的：数据泄漏和模型评价
13.1　Claudia作为数据科学家的知识结构
13.1.1　首席数据科学家的生活
13.1.2　作为一名女数据科学家
13.2　数据挖掘竞赛
13.3　如何成为出色的建模者
13.4　数据泄漏
13.4.1　市场预测
13.4.2　亚马逊案例学习：出手阔绰的顾客
13.4.3　珠宝抽样问题

13.4.4 IBM客户锁定
13.4.5 乳腺癌检测
13.4.6 预测肺炎
13.5 如何避免数据泄漏
13.6 模型评价
13.6.1 准确度重要吗
13.6.2 概率的重要性，不是非0即1
13.7 如何选择算法
13.8 最后一个例子
13.9 临别感言
第 14 章 数据工程：MapReduce、Pregel、Hadoop
14.1 关于David Crawshaw
14.2 思维实验
14.3 MapReduce
14.4 单词频率问题
初涉MapReduce
14.5 其他MapReduce案例
MapReduce不能做什么
14.6 Pregel
14.7 关于Josh Wills
14.8 思维实验
14.9 给数据科学家的话
14.9.1 数据丰富和数据匮乏
14.9.2 设计模型
14.10 算算Hadoop的经济账
14.10.1 Hadoop简介
14.10.2 Cloudera
14.11 Josh的工作流程
14.12 如何开始使用Hadoop
第 15 章 听听学生们怎么说
15.1 重在过程
15.2 不再简单

15.3　援助之手
15.4　殊途同归
15.5　逢山开路，遇水架桥
15.6　作品展示
第 16 章　下一代数据科学家、自大狂和职业道德
16.1　前面都讲了些什么
16.2　什么是数据科学（再问一次）
16.3　谁是下一代的数据科学家
16.3.1　成为解决问题的人
16.3.2　培养软技能
16.3.3　成为提问者
16.4　做一个有道德感的数据科学家
16.5　对于职业生涯的建议

版权声明

© 2014 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2015. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版，2014。

简体中文版由人民邮电出版社出版，2015。英文原版的翻译得到O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc.介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make 杂志，从而成为DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的 知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野并

且切实地按照Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。'回顾过去Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

作者介绍

Rachel Schutt 是美国新闻集团旗下数据科学部门的高级副总裁。她从哥伦比亚大学取得博士学位后，加入谷歌研究院工作了数年。她是哥伦比亚大学统计系的兼职教授，同时也是哥伦比亚大学数据科学及工程研究所的教育委员会发起者之一。她有几个专利正在申请之中，这些专利基于她在谷歌的工作，在那里她设计了算法原型，并且通过建模来理解用户的行为，这些最终都反映在了直接面向用户的产品中。她同时拥有纽约大学的数学硕士学位、斯坦福大学的工程经济系统和运筹学硕士学位，以及密歇根大学的数学学士学位。

Cathy O'Neil 从哈佛大学获得了数学博士学位，她曾经是麻省理工数学系的博士后、巴纳德学院的教授（在那里她发表了大量算术代数几何方面的论文）。随后她转身投入工业界，先是在信贷危机中期加入了D.E. Shaw 公司，担任该公司对冲基金的金融师；然后又加入了RiskMetrics（一家对银行和对冲基金进行风险评估的软件公司）。她现在是纽约初创公司联盟的数据科学家，并且撰写和维护着一个博客mathbabe.org。另外，她还参与了占领华尔街运动。

关于封面图

本书封面上的动物是九带犰狳（Dasypus novemcinctus），是一种广泛分布于中北美及南美洲的哺乳动物。在拉丁文中，novemcinctus 的字面意思是“九条带子”（位于腹部可伸缩甲壳后方），实际上九带犰狳身上的“带子”数量通常为7～11 条。产自南美洲的三带犰狳是唯一一种在遇到危险时可将身体团成球状来保护自身的犰狳，其他种类的犰狳则由于甲壳太多无法做到这点。

犰狳的皮肤最为惹人注目，它的皮肤呈灰褐色、皮革状，由叫作鳞甲的鳞片组成，覆盖了除身下外的其他部分。犰狳长有利爪，善于挖洞，通常在自己的领地内挖好几个洞，并且用臭腺给自己的洞穴做记号。九带犰狳的体重通常为5.5～14 磅，大小和一只大点的家猫差不多。犰狳以昆虫为主食，但是也吃水果、小爬虫和动物的卵。

雌性犰狳一般一胎生四个幼崽，四个幼崽性别都是一样的，这是因为雌性犰狳受精后，受精卵分裂成了四个胚胎。犰狳刚出生时皮肤异常柔软，长大后皮肤慢慢变得坚硬起来。犰狳在出生几小时后即可爬行。

九带犰狳受到惊吓后，可跳起3～4 英尺。虽然这一应激反应能吓走自然界中的食肉动物，但是如果面对一辆行驶而来的汽车，这个反应却是致命的：犰狳会撞上迎面而来的汽车。犰狳和人类还有另外一种不幸的联系，它是唯一已知的麻风病病毒携带者，人类在食用或接触过犰狳后感染上麻风病的消息并不鲜见。

封面图取自英国动物学家乔治· 肖的动物学图谱，由Karen Montgomery 重新着色。

前言

Rachel Schutt

2012年秋天，我在哥伦比亚大学开设了一门新课：数据科学导论。作为一个新兴领域，数据科学在学术界尚未划分为一个独立学科。那么数据科学到底是什么呢？我将这门课的讲义集结成书，试图回答这一问题。

为了帮助读者理解本书及其缘起，我觉得有必要简单介绍一下我自己，和我设计并讲授这门课的初衷。

初衷

简单地说，我期望在我上大学时就有这样的课。但那是20世纪90年代，数据爆炸尚未开始，开设这样一门课也就无从谈起。我本科时主修数学专业，主要是做理论和实证研究。虽然很庆幸这些训练赋予了我严谨解决问题的能力，但同时我也略感遗憾，若当时能再学点实际应用的技巧就更好了。

在从大学毕业到获得统计学博士学位期间，我走了一些弯路，我一直在试图寻找适合自己的研究领域，喜欢探究隐藏在宇宙中的模式，喜欢解答有趣的谜题，希望可以将自己的这些爱好物尽其用。之所以谈起这些，是因为现在很多学生觉得必须先知道自己这辈子到底想要干什么，我做学生时，不可能规划将来要从事数据科学相关的工作，因为那时根本还没有数据科学这样一个领域。因此我建议这些学生，或者其他愿意听我在这儿唠叨的人：大可不必这样。不必现在就规划好未来，走点弯路也没什么，谁知道这一路上你会发现什么呢？我拿到统计学博士学位后，在谷歌工作了几年，在这几年中，数据科学、数据科学家这些术语才在硅谷流行起来。

这个世界有许多问题尚未解决，对于那些拥有量化思维又乐于开动大脑的人来说，在解决问题的过程中充满了机遇。我的目标是帮助学生们成为具有批判性思维的人、能用创新思维去解决问题（甚至是人们尚未发现的问题）的人，对世界充满好奇喜欢问问题的人。若要我去构建一个数学模型，去为治愈癌症贡献一份力量，或者揭示出自闭症的奥秘，或者用来预防恐怖袭击，我或许永远做不到。但我的学生有一天会做到，我教给了他们这些知识，就算完成了自己的使命。写作

此书，使我有机会将毕生所学传播给更多的人，我希望他们能从中得到激励，或者学到一些有用的工具，来让这个世界变得更好，而不是更坏。

建模和数据分析的过程并非彻底地中立，会受到研究者个人价值观的影响。研究的问题是由你来挑选的，研究假设也是你根据模型得出的，度量方法和算法也是由你来设计的。

世界上也并不是所有的问题都需要用数据科学或技术手段来解决，一个好的数据科学家是指他能甄别出哪些问题适合用数据科学解决，构建出对应的数据模型或者编写代码去解决它。但是我相信，在多学科的团队中，如果有一个理解数据、具有量化思维、精通编程的问题解决者（让我们将这种人称为“数据科学家”），这个团队可能会走得更远。

课程的起源

我在2012年3月份提议开设此课，主要原因有三。其中第一个原因最重要，我将会花最大篇幅去阐述。

原因一：我想告诉我的学生业界的数据科学家是怎么工作的，并且让他们掌握一些数据科学家所使用的技术。

在为Google+工作时，我所在的数据科学团队由一群身怀绝技的博士组成，其中有学社会学的、学工程的、学物理的和学计算机的，而我是统计学专业的。我们隶属于一个更大的团队，这个团队有很多天才的数据工程师，他们实现数据管道、基础架构、分析面板和一些实验性质的架构（用来做A/B测试）。我们的团队架构是扁平化的，我们有海量的数据，每个人都是各自领域的专家，我们精诚合作，做出了很多不可思议的事，包括建立预测模型、实现算法原型、揭示出隐藏在数据背后的模式，这些对我们的产品影响深远。

以数据为基础，我们为领导层的决策提供真知灼见；分析因果关系，我们发展出了新的方法论。这些全仰仗世界一流的工程师和技术设备。每个人都为团队引入了专家级的技能，包括编码、软件工程、统计学、数学、机器学习、通信、可视化、探索性数据分析（EDA）等，还有对社交网络和社交空间的数据的敏感直觉和专业知识。

要知道，没有人是全知全能的，但集合所有人的智慧，我们就做到“无所不能”。我们认识到了每种技能的价值，因此就成功了。我们

的共同点是守信，对解决有趣的问题充满好奇心，对待新的科学发现既保有适度的怀疑又充满激情。我们喜爱这项工作，对数据背后的模式充满了好奇。

我居住在纽约，希望把我在谷歌公司的工作经验传授给哥伦比亚大学的学生们，我相信他们需要这个，而且，我也喜欢教学。我想把我从工作中学到的东西教给他们。另外，我知道纽约的技术圈里有一个新兴的数据科学家社区，我也希望学生们能从他们身上汲取知识。

因此，这门课程常会邀请业界或学术界的数据科学家来做客座演讲。每位嘉宾所专长的技能和领域都不尽相同。我希望通过这样一种多样性的组合，让学生们对数据科学有一个更全面的认识。

原因二：数据科学有希望成为一门极具研究价值、意义深远的学科，它会影响到人们生活的方方面面。为此，哥伦比亚大学和纽约市市长布隆伯格先生在2012年7月宣布成立了一个数据科学与工程研究所。开设这门课是在尝试发展数据科学的理论，我希望让数据科学成为一门真正的科学。

原因三：我时常听到业界的数据科学家说，在脱离实践的课堂上是无法真正教授数据科学的，我想挑战一下这种言论。我一直将我的课堂视作数据科学家的孵化器，而我的学生也确实表现出色，他们将会成为数据科学界冉冉升起的新星。事实上，本书其中一章内容就是由我的学生们贡献的。

本书的起源

如果不是遇到了Cathy O'Neil，我的教学笔记也不会集结成书。她是一位数学家，后来转型为数据科学家，她的个人博客mathbabe.org很受欢迎，在博客中的“关于自己”部分，她说自己一直在期待下面这个问题能有更好的答案：非理论派的数学家能做些什么以让这个世界变得更加美好？我向大学提议开设数据科学导论这门课程时，恰好认识了Cathy，那时她正在一个初创公司工作，职位是数据科学家。对于我开课的尝试，她十分支持。她还提出亲自过来听课，并在博客上同步直播我的授课内容。鉴于我性格比较内向低调，起先我并不喜欢这么做，后来Cathy说服了我。她说这与商业广告的肆意炒作截然不同，这是一个绝好的机会，借此可以将“数据科学”的概念向大众普及。

我在哥伦比亚大学上的每一节课，Cathy都会坐在第一排，并不时提

出问题。她后来还受邀作为这门课的客座嘉宾给同学们上了一课（见第6章）。除了将我的讲义发布到博客上，Cathy还对授课内容贡献甚巨，比如，她提醒我们数据建模过程中存在一些道德伦理方面的考量。此外，她鼓励我也同步开设一个博客（http://columbiadatascience.com/blog/），用来和学生们做直接交流。我在上面也会总结自己的教学经验，这或许会帮到其他教授。Cathy博客中所有关于我授课内容的条目，再加上我博客中的部分内容，构成了本书的原始素材，我们在这一基础上修改加工，再集合一些其他资料，终成此书。

本书内容

本书既介绍实践应用，也提出理论规范。一方面，本书介绍了一些业内顶尖数据科学家的日常工作内容，带大家看看他们在实践中如何应用数据科学知识，借此管中窥豹，了解这一学科目前的应用现状。另一方面，我们还将从学术角度去定义数据科学的研究范畴。

这不是一本关于机器学习的教科书。恰恰相反，本书会多角度全方位、深入地介绍数据科学。它是对现有数据学科领域的纵览，试图为这一学科勾勒出一幅全景图。因此，在选择案例时，我们会更注重广度而非深度。

希望本书能够被那些善待它的人充分利用，举一反三，去解决那些重要的问题。

这门课在哥伦比亚大学讲完后，我听到了这样的评价：它是一门从人文主义角度、全面讲解数据科学的课程。我们不仅关注工具、数学、模型、算法和代码，同时也很关注上述过程中的人性化考量。关于什么是人文主义者，我很喜欢如下的定义：“他十分关心人类的福祉，尊重个人的价值观，并且注重维护个体尊严。”如何在数据科学中体现人文主义？你在建模和设计算法时，认识到你作为个人所应起到的作用，想想哪些东西是人所具备而电脑不具备的，比如基于道德的判断；向世界公布一种新的统计模型前，想想会为他人的生活带来什么样的影响。

组织结构

本书的组织结构遵循我在哥伦比亚大学的数据科学导论课程，在第1章，我们将会回答“什么是数据科学”这个核心问题，同时介绍数据科

学工作流程，这是全书组织结构的纲领。第2章和第3章对统计模型和机器学习算法做一概览，它们是后续章节的基础。第4章到第6章，以及第8章将会针对特定案例深入学习一些模型和算法。第7章讲述如何从数据中提取有效信息以及在模型中创建统计变量。第9章和第10章将深入介绍一些传统学术界很少涉足的内容（当然现在情况有所改善）：数据可视化和社交网络。第11章和第12章将从预测模型转而介绍因果分析。第13章和第14章介绍数据预处理以及工程方法。第15章是我的学生们讲述他们的故事——他们是怎样学习数据科学的。第16章展望数据科学未来的发展。

阅读须知

阅读本书时最好从前往后依序阅读，这样更便于理解，因为不少概念都是一环扣一环的。如果你的统计和概率背景不强，或者从前没有编过程，那么阅读本书的同时，如能阅读本前言末尾附带的补充材料以查漏补缺，效果将会更好。全书为大家推荐了很多补充材料，当你阅读某个章节感到困难时，这或许由于你缺失某些背景知识，或许由于我们的讲解不够清晰，这时你都可以求助于这些补充材料，厘清概念。

书中的代码

本书不是一本手册，书中代码仅供示范，有时需要读者自己去亲手实践实现某些算法，以期对其中的概念理解得更加深入。

目标读者

虽然媒体把数据科学和数据科学家渲染成摇滚巨星一般，但别怕，数据分析学科的门槛并没有那么高。你是否是金融工程师并不重要，只要你热爱解决问题，热爱寻找数据背后的模式，那么数据科学就绝对是你的菜。

具备各种专业背景的读者均可阅读本书，我们希望每位读者都能各取所需，读出自己心中的“哈姆雷特”。

- 有经验的数据科学家能从一个全新的角度认识自己和自己从事的行业。
- 统计学家能从中了解到统计学和数据科学的关系。也许他们仍然坚持过去的老态度：什么数据科学，这就是统计学嘛。我们

也不介意这样的争论。
- 金融、数学、物理等学科的博士们，若有意转行研究数据科学，或者只是希望学习一些数据科学技能，都能从本书中了解到他们需要学些什么。
- 学生或从未接触过数据科学的人，会觉得被直接扔进了该领域的深水区。如果不能一下子理解书中内容，请不要害怕，这是必经之路。
- 对于那些从未用R或Python写过程序的读者，我们推荐一本书：*The Art of R Programming* ，Norman Matloff著（No Starch Press），在哥伦比亚大学选修过该课程的学生们也从实验讲师Jared Lander的讲解中获益良多，他的新书*R for Everyone: Advanced Analytics and Graphics*（Addison-Wesley）已于2013年12月上市。书中所有练习也可以使用Python的各种工具包实现。
- 对于那些完全没有编程经验的读者，上述建议也适用，但可能需要先阅读一些基础的编程书籍，如下两本书就是不错的选择： Mark Lutz和David Ascher的*Learning Python*（O'Reilly），以及Wes McKinney的*Python for Data Analysis*（O'Reilly）。

基础知识要求

我们假设本书读者学过线性代数、概率论和统计学，还有一些编程经验。不过，没有学过也没关系，我们试图让本书在内容上能够尽可能涵括所有内容。当然，如果读者在某方面知识欠缺，大可以针对不足自行寻找其他补充阅读材料。我们在全书中也都尽可能向读者提示哪些补充阅读材料能让你更深刻地理解相关问题。

补充阅读

数据科学作为一个新兴领域，植根于其他学科：统计推断、算法、统计模型、机器学习、实验设计、优化理论、概率论、人工智能、数据可视化和探索性数据分析等。每门学科都值得花好几门课或好几本书专门讲解，这正是写作本书面临的一个极大挑战，因此我们将这些补充阅读材料附在后面，希望读者在需要时可以参考。

数学

- *Linear Algebra and Its Applications*，Gilbert Strang著

（Cengage Learning）
- *Convex Optimization*，Stephen Boyd和Lieven Vendenberghe著（Cambridge University Press）
- *A First Course in Probability*（Pearson）、*Introduction to Probability Models*（Academic Press），Sheldon Ross著

编程

- *R in a Nutshell*，Joseph Adler著（O'Reilly）
- *Learning Python*，Mark Lutz和David Ascher著（O'Reilly）
- *R for Everyone: Advanced Analytics and Graphics*，Jared Lander著（Addison-Wesley）
- *The Art of R Programming: A Tour of Statistical Software Design*，Norman Matloff著（No Starch Press）
- *Python for Data Analysis*，Wes McKinney著（O'Reilly）

数据分析与统计推断

- *Statistical Inference*，George Casella和Roger L. Berger著（Cengage Learning）
- *Bayesian Data Analysis*，Andrew Gelman等著（Chapman & Hall）
- *Data Analysis Using Regression and Multilevel/Hierarchical Models*，Andrew Gelman和Jennifer Hill著（Cambridge University Press）
- *Advanced Data Analysis from an Elementary Point of View*（http://goo.gl/udICRX），Cosma Shalizi著（Cambridge University Press）
- *The Elements of Statistical Learning: Data Mining, Inference and Prediction*，Trevor Hastie、Robert Tibshirani和Jerome Friedman著（Springer）

人工智能和机器学习

- *Pattern Recognition and Machine Learning*，Christopher Bishop著（Springer）
- *Bayesian Reasoning and Machine Learning*，David Barber著（Cambridge University Press）
- *Programming Collective Intelligence*，Toby Segaran著（O'Reilly）
- *Artificial Intelligence: A Modern Approach*，Stuart Russell和

Peter Norvig著（Prentice Hall）
- *Foundations of Machine Learning*，Mehryar Mohri、Afshin Rostamizadeh和Ameet Talwalkar著（MIT Press）
- *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*，Ethem Alpaydim著（MIT Press）

实验设计

- *Field Experiments*，Alan S. Gerber和Donald P. Green著（Norton）
- *Statistics for Experimenters: Design, Innovation, and Discovery*，George E. P. Box等著（Wiley-Interscience）

可视化

- *The Elements of Graphing Data*，William Cleveland著（Hobart Press）
- *Visualize This: The FlowingData Guide to Design*，*Visualization, and Statistics*，Nathan Yau著（Wiley）

关于本书其他贡献者

我邀请了很多嘉宾来我的数据科学导论课上做讲座，他们有的来自初创公司、科技企业，当然还有人就是我所在大学的教授。书中多章的内容都是在这些讲座的基础上写成的。虽然他们没有执笔写作此书，但书中很多内容及想法皆来自他们，此外，他们还帮助我审阅了各自讲座所对应章中的内容，并给出了很好的建议，对此我们深表谢意。没有他们我的数据导论课就不会成功，没有他们也不会有这本书，他们是我在数据科学领域的楷模。

排版约定

本书使用的排版约定如下。

- 楷体
  表示新的术语。

- 等宽字体（`Constant width`）
  表示程序片段，也用于在正文中表示程序中使用的变量、函数名、命令行代码、环境变量、语句和关键词等代码文本。

- 加粗的等宽字体（**`Constant width bold`**）
  表示应该由用户输入的命令或者其他文本。

- 斜体的等宽字体（*`Constant width italic`*）
  表示应该由用户输入的值或根据上下文决定的值替换的文本。


这个图标代表小窍门、建议或说明。


这个图标代表警告信息。

使用代码示例

你可以在这里下载本书随附的资料（数据集、练习题等）：https://github.com/oreillymedia/doing_data_science。

本书旨在帮助读者解决实际问题。一般来说，对于书中提到的代码，也许你需要在自己的程序或文档中用到，但除非大段大段地使用，否则你不必与我们联系取得授权。因此，用本书中的几段代码写成一个程序不用向我们申请许可。但是，销售或者传播O'Reilly图书随附代码的光盘必须事先获得授权。引用书中的代码来回答问题你也无需我们授权，将大段的示例代码整合到你自己的产品文档中则必须经过许可。

使用我们的代码时，希望你能标明它的出处。出处一般要包含书名、作者、出版社和ISBN，例如：*Doing Data Science* by Rachel Schutt and Cathy O'Neil (O'Reilly). Copyright 2014 Rachel Schutt and Cathy O'Neil, 978-1-449-35865-5。

如果还有其他使用代码的情形需要与我们沟通，可以随时与我们联系：permissions@oreilly.com

Safari® Books Online



Safari Books Online（http://www.safaribooksonline.com）是按需获取的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

技术专家、软件开发人员、Web设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将Safari Books Online视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解Safari Books Online的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：
O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：
北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：http://oreil.ly/doing_data_science。

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多O'Reilly图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在Facebook的地址如下： http://facebook.com/oreilly

请关注我们的Twitter动态： http://twitter.com/oreillymedia

我们的YouTube视频地址如下： http://www.youtube.com/oreillymedia

致谢

感谢在谷歌的同事：David Huffaker、Makoto Uchida、Andrew Tomkins、Abhijit Bose、Daryl Pregibon、Diane Lambert、Josh Wills、David Crawshaw、David Gibson、Corinna Cortes、Zach Yeskel和Gueorgi Kossinetts。另外，还要感谢在哥伦比亚大学统计系的朋友们：Andrew Gelman和David Madigan，还有课程的实验指导者和助教Jared Lander和Ben Reddy。

最后，也最真诚地感谢家人和朋友，他们提供了无尽的支持和爱，他们是：Eran Goldshtein、Barbara和Schutt、Becky、Susie和Alex、Nick、Lilah、Belle、Shahed，以及Feeneys一家。

——Rachel Schutt

感谢朋友与家人，尤其是可爱的儿子和丈夫，他们允许我每周为这门课程写一次博客。

——Cathy O'Neil

我们还要共同对以下人士和单位致谢。

- 在Cathy家参加聚会的那些志同道合的朋友：Chris Wiggins、David Madigan、Mark Hansen、Jake Hofman、Ori Stitelman和Brian Dalessandro。
- 编辑Courtney Nash和Mike Loukides。
- IMA User级建模会议的参与者及组织者， 很多关于本书的基本构思即来源于那里。

- 学生们！
- Coppelia餐馆，Cathy和Rachel经常碰头和吃早餐的地方。

最后，我们将谢意诚敬给Johnson实验室的John Johnson和David Park，感谢他们的慷慨大方，提供给我充足的时间用来写作此书。

第 1 章　简介：什么是数据科学

过去几年，“数据科学”和“大数据”的概念被媒体炒得热火朝天。对于这种现象，人们一开始难免疑惑，甚至怀疑。事实上，这就是Cathy和我当时的反应。

对于这些概念，Cathy和我在很长一段时间里都感到迷茫，直到我们俩相识。我们一般会在星期三共进早餐，每当谈起这种现象，都有一种不安的感觉，总觉得在这喧嚣背后确然有一股新潮流在涌现，这股潮流或许是意义深远的，代表着我们整个文化范式在数据的影响下都会产生深刻的改变。Cathy和我都是干这行的，觉得应该发挥我们的强项，去探索这些现象背后的原因，而不是置之不理。

在深入探索之前，我们有必要先介绍一下媒体所炒作的大数据时代，也许你和我们一样，也认为那些概念难以理解、语焉不详。然后，本章会进一步讲解我们是如何拨开迷雾发现背后的真相，以至于Rachel决定在哥伦比亚大学开设数据科学导论课程，而Cathy则在她的博客上同步记录该课程的内容，乃至上述所有内容终于结集成书送到你手中。

1.1　大数据和数据科学的喧嚣

让我们抛开炒作，因为很多人可能和我们一样，都对数据科学心存怀疑。之所以一上来就讲这些，是想让你知道：我们也和你一样！假如你也心存疑虑，说明你也很可能会贡献一份力量，推动数据科学的健康发展，使其对社会产生积极的影响，也使数据科学这门学科趋于正统，在众多学科中能占有一席之地。

让我们先来细数大数据和数据科学之所以这样让人如坠云里雾里的原因。

1. 大多数基本的术语都缺乏严格定义。究竟什么是大数据？数据科学又是什么意思？大数据和数据科学之间有什么关系？数据科学就是关于大数据的科学吗？只有像谷歌和Facebook这样的高科技企业才用得到数据科学吗？为什么有人认为大数据是一个交叉学科（比如天文学、金融学、科技等），但数据科学却只是科技界的事儿？大数据，多大才是大？这些术语及概念如此含混不清，简直毫无意义。

2. 对于数据科学领域的研究者，不管是在学术界还是工业界，公众都缺乏敬意。事实上，他们在这一领域内辛勤工作了很多年，而这些工作是继承了各个领域的前辈们数十年甚至数百年的工作成果，这些领域包括统计学、计算机科学、数学、工程学以及其他学科。而媒体传播给公众的信息却是这样的：机器学习算法是上个礼拜才发明出来的，谷歌出现之前都不存在所谓的大数据。这简直荒谬，很多正在使用的方法和技术，还有我们面临的挑战，都不过是在过去已有的方法、技术和挑战上演变而来的。我们并不否认新事物和新技术的出现，只是觉得应该对历史和前人的研究成果保持必要的敬意。

3. 媒体疯了。人们将各种各样的桂冠加诸数据科学家的头上，人们形容他们是掌握了宇宙奥秘的魔法师，其疯狂程度堪比金融危机之前。天花乱坠的宣传很容易掩盖真相、歪曲事实。这些宣传的噪声越多，真正有效的信息就越少。因此，若“大数据”被媒体吹得越久，公众越容易被误导，越难获知这一概念背后真正有益于社会的一面（如果有的话）。

4. 统计学家觉得他们正在干的事就是数据科学。换句话说，这本来就是他们的饭碗。亲爱的读者们，请设身处地替统计学家们想想，有人抢自己的饭碗是什么感受。媒体也常常将数据科学轻描淡写为统计学和机器学习在科技界的简单应用。我们会在书中阐明，不是说将统计学和机器学习这些“旧酒”装进新瓶里，就叫作数据科学。它绝对有资格作为一个独立的学科存在。

5. 所有自称为科学的都不是真正的科学。这句话或许有些道理，但不代表数据科学这一术语毫无意义，它代表的可能不是科学，而是某种技术。

1.2 冲出迷雾

Rachel取得统计学博士学位到她在谷歌工作的这段经历，或许能帮我们解答一些疑惑，她说：

> 进入谷歌之后，我很快就意识到工作中用到的东西和我读统计学博士学位时学到的东西差别很大。并不是说我的统计学知识毫无用武之地，相反，我在学校学到的东西为我思考问题提供了一个框架，统计学的很多知识都为我的日常工作提供了坚实的理论和实践基础。

> 在谷歌工作期间，我发现必须掌握很多在学校没学到的东西，比如计算、编程、数据可视化技能和许多领域知识。这种经验既特殊又普遍，我拥有统计背景，因此需要补充前面提到过的那些知识，而若换作一位计算机、社会学或者物理学背景的人，他们也需要根据自己的知识缺陷去补充相应的知识。每个人都拥有自己独特的知识结构，重要的是大家能够紧密合作，取长补短，组成一个团队去解决数据问题。

一般人对上述故事肯定会有这样一种想法：你走上工作岗位后就会发现，在学校学到的知识，远远不能满足实际工作的需要。因此，本书中教授的统计学知识与业界所应用的统计学方法，肯定也是不尽相同的。对此，我们有一些自己的看法。

- 为什么学校里的统计要和工业界的统计如此不同？为什么很多学校的课程要和现实如此脱节？
- 这种差异不仅存在于学校里的统计和工业界的统计之间。很多数据科学家的一个共同感受是，工作时他们需要接触更多的知识、方法论和工序（详见第2章），而这些东西都是以统计学和计算机科学为基础的。

抛却这些媒体给予数据科学的光环，只有一件事是实在的：数据科学是一个新生事物。它刚刚诞生，却被赋予了太多荣耀，使人们对其充满了很多不切实际的幻想，而幻想最终是会破灭的。我们要保护数据科学，过分吹捧可能会让这个新兴领域过早夭折。

Rachel决定去研究数据科学这一文化现象，她想了解其他人对数据科学的感受。她开始和谷歌的人接触，和很多创业公司和高科技公司的人接触，和大学（特别是统计系）里的老师们接触。

从这些接触中，Rachel觉得数据科学的轮廓渐渐清晰起来，她进一步深入，决定在哥伦比亚大学开设一门数据科学导论课程，与此同时Cathy在博客上连载了该课程的讲义。我们期望在这门课程结束时，我们和学生们能对数据科学的本质有一个清晰的理解。现在我们把课程的内容集结成书，也是希望帮助更多的人去了解数据科学。

1.3　为什么是现在

现在，数据充斥在我们生活的方方面面。网络购物、网上通信、浏览新闻、收听在线音乐、搜索信息，或在网上表达观点，这些行为都会

被记录。同时，我们拥有充足且廉价的计算能力。有数据，有计算能力，这为从事数据科学提供了良好的环境。

大家都知道，线上数据的收集正在经历一场革命（稍后会详细介绍），但他们所不知道的是，离线数据的采集同样也在革新。人们的日常行为也被“数据化”了。将二者结合起来，我们可以深入研究人类的行为，甚至从更高的物种角度，来研究人类行为区别于其他物种的特殊性。

数据也不局限于互联网产生的数据，金融、医疗、制药、生物信息、公共福利、政府、教育、零售等行业都会产生大量的数据，数据在各行各业的影响力在与日俱增。部分行业所储存的信息达到了“大数据”的程度（详见第2章），而另一些行业的信息量则没有那么多。

数据科学这一课题变得日益有趣（或提出了新的挑战），这不仅仅是因为数据的体量增大，更多的是因为数据本身（很多时候是实时数据）成了构建数据产品的关键要素。在互联网上，有亚马逊的推荐系统、Facebook的朋友推荐系统，还有其他的图书、电影、音乐等推荐系统；在金融业，有信用评级系统、交易算法和模型；在教育领域，可以实现教育对学生的量身定制，比如现在的网络培训公司Knewton和网络大学Khan Academy；在政府机构中，这意味着以数据为基础去制定公共政策。

我们正在见证一个时代的开始，这个时代是一个巨大的、充斥着人文特色的反馈环：我们的行为会改变产品，产品又反过来影响我们的行为。技术使这一切成为可能，我们拥有处理大数据的基础架构、更大的内存和更快的网络，而且社会公众也日渐认同技术是生活中必不可少的组成部分。在十年前，这一切我们还不敢想象。

由于这种基于反馈的循环对社会变革将产生不可小觑的影响力，我们认为，有必要认真考虑如何确保这种循环的良性运行，尤其是直接参与这一过程的人员，在实践中应保持哪些道德准则、应负何种责任。本书的目的之一就是针对这些话题开展一些抛砖引玉的探讨。

数据化

*Foreign Affairs*杂志在2013年5/6月期刊上发表了一篇由库克耶和迈尔-舍恩伯格共同撰写的文章“The Rise of Big Data”（大数据的崛起）。该文谈到了数据化的概念，以朋友之间的关系为例，他们将对朋友的喜欢程度转化为数值，这些数据被存储起来，用于日后研究，或者出售。将问题数据化，这是我们人类处理问题时经常采用的一种

方式，不管是线上还是线下。

在文章中，数据化被定义为一种处理流程，它将生活的方方面面转化为数据。比如，谷歌眼镜将其所视范围内的景象转化成数据，Twitter将人们偶尔产生的想法转化成数据，LinkedIn将职业社交网络转化成数据。

数据化是一个很有趣的概念，我们在重视它的同时，必须尊重他人的意愿——是否自愿与人们分享自己的数据。比如，在网上为某个人或某件东西"点赞"时，人们要么是故意让自己的行为"被数据化"，要么最低限度上也清楚自己的行为会被记录下来。但有时却不然，我们只是随意浏览一些网站，我们的行为却被网站上的cookie记录下来；我们走进商店，或者只是走在大街上，会被各种传感器、摄像头监测，或者被谷歌眼镜拍摄，我们的行为被作为数据存储下来，而这种数据化并非出于我们的意愿。

数据化无所不在，从作为实验对象参与到社交媒体实验中，到接受全面调查，再到被人秘密跟踪，这些都是被数据化的典型案例，它们代表了数据化过程中个人意愿从高到低的各种情形，但其产生的结果却远不能如此简单地划分概括。

在文章中他们又说：

> "一旦我们可将问题数据化，就能改变人们的意图，并在这些信息基础上产生新价值。"

本书会不时提出这样一个问题：究竟谁才算"我们"？"新的价值"是什么？在他们的文章中，"我们"显然指那些模型和企业，他们引导用户购买更多的产品，赚取更多的钱，"新价值"则指那些能提高效率的方法，比如通过自动化等。

如果将视野放得更大，将这里的"我们"指代更广泛的人类，那就有点逆潮流而行的意思了。在面对数据化的大潮时，我们或许会有所保留。

1.4　数据科学的现状和历史

那么，到底什么是数据科学？它是一门新生事物，还是统计学的旧瓶子里装了新酒？它是纯粹的炒作，还是确有其事？如果它是一门实实在在的新兴学科，它的意义何在？

让我们先上网看看业界关于这一问题的讨论，这不一定能直接回答我们的问题，但听听别人怎么说总是有益的。2010年，Quora网站有一个关于“什么是数据科学”的提问，Metamarket公司的CEO Mike Driscoll的回答如下。

> 研究数据科学，一方面需要如极客那般刻苦钻研，一方面需要像统计学家那样拥有完美的理论。
>
> 数据科学家不仅仅是极客——极客只关心如何调试一行Bash脚本或Pig脚本，没人会在意非欧氏距离矩阵。
>
> 数据科学家也不仅仅是统计学家——后者只关注如何完成一个理论的证明或构建出一个完美的模型，很少有人会使用R语言将数据文件读入系统，从而进行后续的分析。
>
> 数据科学是一门关于数据的工程，它需要同时具备理论基础和工程经验，需要掌握各种工具的用法。

Driscoll随后引用了2010年Drew Conway的韦恩图来说明研究数据科学需要的技能，如图1-1所示：

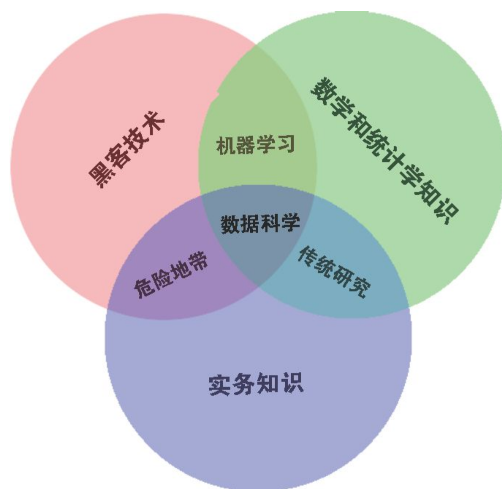


图1-1：Drew Conway的数据科学韦恩图

Driscoll还引用了Nathan Yau 2009年的一个关于“数据科学家正在涌现”的帖子，其中介绍了数据科学家们应该具备的各种技能：

- 统计学（做传统分析时需要的技能）
- 数据处理（解析、提取和格式化数据）
- 可视化（图表、工具等）

但是先别忙，如此说来，数据科学就是这些技术的一个简单的组合吗？抑或是诸如统计学、机器学习等学科的一个逻辑上的扩展？

Cosma Shalizi1和Cathy2分别就统计学家和数据科学家的区别这一问题发表了很多看法。Cosma认为，任何一个够格的统计部门都在从事数据科学的工作，数据科学只不过是统计学换了个新说法。

> 1http://goo.gl/SO7ceN和http://goo.gl/pXg1fU。

> 2http://goo.gl/F4K4hE和http://goo.gl/X9Bmxj。

持此观点的还有ASA主席Nancy Geller，她在2011年发表的一篇文章“Don't shun the 'S' word”中说：

> 我们要告诉人们：是统计学家揭示出数据的含义。在21世纪，各行各业都涌现出了海量的数据，无论是科学、工程还是医学，从文学史到动物学，人们在处理这些数据时都应用了统计学技术。这种数据大爆炸，为统计学者提出了源源不断的研究课题，因此在这个时代从事统计学工作是一件相当令人兴奋的事。

Nancy以为用“从文学史（Art history）到动物学（Zoology）”这种说法，就可以巧妙地暗喻“从头到尾”2的概念，代表了数据科学的应用无处不在。但她这种说法却是搬起石头砸了自己的脚，因为她所罗列的全是学术界的例子，恰恰不包含高新技术企业，而业界才是数据爆炸式增长最迅猛的地方，数据科学也是在这些高新技术企业里得到了长足的发展。在企业中，会有数据科学家的职位，但这一称号在学术界还很少见到（或许这点会慢慢改变）。

> 2from a to z，意即“完全、彻底”。——编者注

不久前DJ Patil和Jeff Hammerbacher讲述了2008年，他们是如何分别在LinkedIn和Facebook上定义了“数据科学家”这一称谓的。2008年，“数据科学家”成为一个职位，出现在这两家公司的招聘信息里（维基百科于2012年增加了数据科学的相关词条）。

当一组技术在谷歌得到追捧，而且这种势头蔓延到硅谷的其他高科技公司时，一个新的职位就会出现，而当这成为常态，人们就需要给它一个全新的名字，比如数据科学家。当这个新名字声名远播，所有人都希望自己成为一名数据科学家。《哈佛商业评论》（*Harvard Business Review*）把数据科学家誉为“21世纪最性感的工作”，这无疑是火上浇油。

> 社会学家在数据科学中的角色
>
> LinkedIn和Facebook都是做社交网络的公司，他们所谓的数据科学家经常是对统计学家、软件工程师和社会学家的统称。这很好理解，因为他们的产品就是社交工具，主要处理的内容是个人（用户）行为。但是根据Drew Conway的韦恩图，数据科学所研究的问题经常是跨领域的，也就是需要大量的“实务知识”（见图1-1）。
>
> 也就是说，数据科学家要用到哪些“实务知识”，就要具体问题具体分析了。如果你要解决的是跟社交网络相关的问题，比如说“好友推荐”“可能认识的人”以及“用户分类”等，那一定要把社会学家拉进来。社会学家大多都擅于提问，他们也热爱调查研究，如果他们再会定量分析和编程，肯定会成为优秀的数据科学家。
>
> 由于“历史”的原因（其实不过是2008年的事），人们认为数据科学家的工作只是负责分析在线用户的行为数据。而现在兴起了一个全新研究领域，它被称作“计算社会科学”，我们可以将其视作数据科学的一个子集。

让我们回到更早的2001年，当时William Cleveland写了一篇关于数据科学的文章“Data Science：An action plan to expand the field of statistics”。

那么，是先有的数据科学还是先有的数据科学家？

这就引出了一系列问题：我们能通过数据科学家的工作来定义数据科学吗？谁有资格定义这个全新的学科？媒体制造了很多关于数据科学的时髦用语，但他们有资格定义吗？我们需要依赖于这些自诩的数据科学家吗？到底有没有这样一个权威机构？我们暂且不予回答。

数据科学的职位

在布隆伯格的帮助下，哥伦比亚大学决定成立一个新的研究所用于数据科学和工程方面的研究。据我们上一次统计，仅在纽约就有465个数据科学的就业机会。即使数据科学还算不上一个真正的领域，但它已经在产生实实在在的工作职位了。

在这些招聘职位的描述中我们发现，数据科学家被要求具备计算机科学、统计学、传播学、数据可视化等领域的知识，还要是一个“通才”。但事实上，没有人能如此面面俱到，因此组建一个具备多种技能的团队更为可行。通过组建团队让不同领域的专家通力合作，这基本上就可以达到数据科学家的“通才”的要求了。我们先来看看现今的数据科学家需要具备哪些素质。

1.5　数据科学的知识结构

在数据科学导论的课堂上，Rachel发给每个学生一张卡片，让他们根据在如下领域的技能水平填写自己的知识结构：

- 计算机科学
- 数学
- 统计学
- 机器学习
- 某一领域的专业知识
- 沟通和演讲的技巧
- 数据可视化

图1-2显示了Rachel在数据科学方面的知识结构。

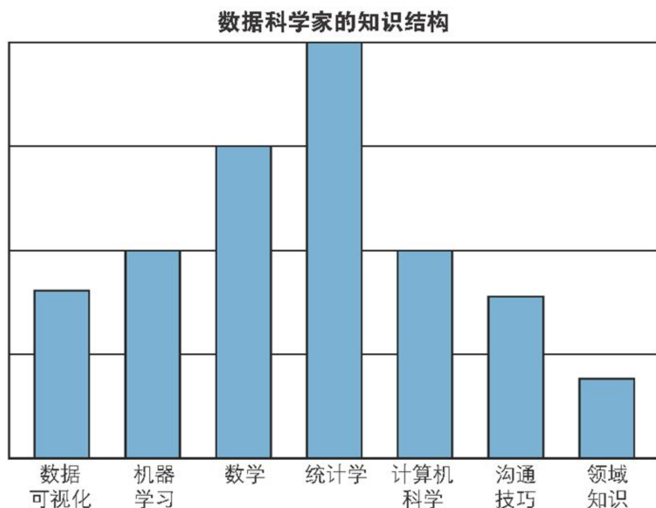


图1-2：Rachel的数据科学知识结构，她试图以此图描绘一个数据科学家应该具备的技能。她希望学生们和客座讲师们都能绘制自己的图谱，并且通过这样的自我检视来发现知识结构中存在的不足

我们把这些卡片钉在黑板上审视一番，发现个体之间技能上的差异还是很大的，这点让我们很满意。比如说，学生中很多都拥有社会学的教育背景。

你在数据科学方面的知识结构是什么样子的呢？你想它在几个月后变成什么样呢？几年后呢？

像我们早先提到的那样，最佳选择可能就是让拥有不同技能的人组成团队进行数据科学方面的工作，因为没人可以掌握所有的知识。于是，我们开始思考，相较于定义“数据科学家”，是否定义“数据科学团队”更有意义？图1-3定义了一个数据科学团队：

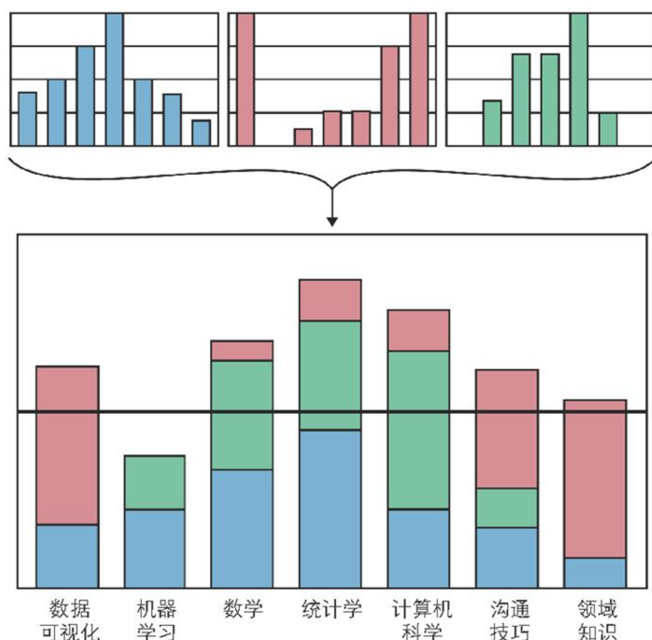


图1-3：数据科学团队的知识结构由每个成员的知识结构叠加而来，在组建团队时，要让团队技能与所解决的问题大致匹配

1.6　思维实验：元定义

每一节课上都会有一个“思维实验”的环节，我们把学生分成小组来讨论问题。很多问题都是开放性的，我们只想借此引发学生就数据科学的相关问题展开更广泛的讨论。在第一节课上，我们的思维实验是：可以通过数据科学的手段来定义数据科学吗？

通过分组讨论，同学们提出了一些有意思的想法。

> 使用文本挖据模型
> 首先在谷歌上搜索“data science”（数据科学），对搜索结果进行文本挖掘。但在语言的选用上，使用者和从业者的原则是截然不同的。作为使用者，我们会采用大众的定义（所谓大众的定义，即通过谷歌搜索得来的结果）。而对于从业者而言，若能引用权威渠道（比如《牛津英语词典》）的说法来定义数据科学会更严谨一些，但可惜的是这些词汇恐怕尚未收入其

中，而且我们也没有耐心去等待了。所以，我们不得不承认，数据科学的定义包罗万象，但目前没有一种定义能让各方都满意。

使用聚类算法
何不考虑数据科学的从业者，看看他们是怎么形容自己的工作的（也许最开始是“单词云”的形式）？然后，我们再看看其他行业的从业者，比如统计学家、物理学家、经济学家，看看他们又是怎么形容自己的工作的。然后，我们使用聚类算法（将在第3章用到）或者其他模型，看看根据对工作内容的描述，是否可以预测出其从事的行业。

作为对比，让我们来看看Harlan Harris是如何进行调查，使用聚类算法定义数据科学的子领域的，如图1-4所示：

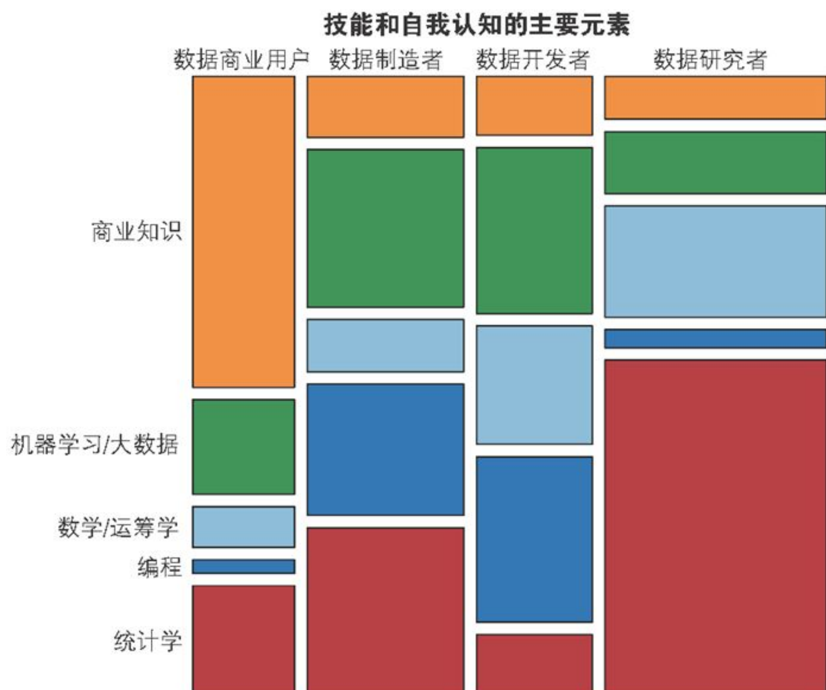


图1-4：此图使用聚类算法描述了数据科学的子领域，源自Harlan Harris、Sean Murphy和Marck Vaisman基于2012年年中对数百名数据科学从业者的调查所著的*Analyzing the Analyzers*（O'Reilly）

1.7 什么是数据科学家

也许定义数据科学最具体的方式是看它如何被使用，比如雇主们都花钱让数据科学家去做哪些工作。以此为目的，我们将会具体说说数据科学家究竟都在干什么，不过我们先来看看学术界。

1.7.1 学术界对数据科学家的定义

在学术界，现在还没人称自己是数据科学家，除非他们工作于某大学的“数据科学研究所”，或者在申请数据科学研究的经费，这时，他们才勉强将数据科学家作为自己的第二称谓。

不如我们问另外一个问题：在学术界，哪些人打算成为数据科学家？在哥伦比亚大学的数据科学导论课学习的有60名学生，Rachel打算开设此课时，估计这门课的学生主要来自统计学系、应用数学系和计算机科学系。事实上，后来学生背景的多样化大大超出了她的预想：除过上述三个领域之外，她的学生还有来自社会学、新闻学、政治学、生物医学信息学、建筑学、环境工程、纯数学和商业学院的，此外还有来自纽约市政府机构以及关注社会福利的非盈利性机构人员。其中不乏一些已经在从事数据科学工作的人。他们都很希望能使用数据去解决一些重要的问题，通常这些重要问题具有重要的社会价值。

想要使“数据科学”在学术界立得住脚，其所要研究的领域应该有更规范的定义。值得一提的是，现在数据科学领域已经有很多可以转化成博士论文的研究课题。

让我们试着定义数据科学家：一个学术界的数据科学家首先是个科学家，他接受了任何其他学科的训练（从社会学到生物学等各种学科），还要同大量的数据打交道，不管这些数据的结构、规模以及复杂程度如何，他都能挖掘出数据背后的意义，从而解决现实世界中的问题。

上述例子说明，在不同的学术领域人们面临的计算和深度数据问题都存在较大的共性。若不同机构的研究者通力合作，他们就可以解决各个领域的现实问题。

1.7.2 工业界对数据科学家的定义

那么工业界的数据科学家又在做些什么？这取决于数据科学家的资深程度以及是否将数据科学特别限定在互联网领域。数据科学家这个职

位也不是只有科技界才有，但是数据科学这个词的确源自科技界，为了避免混淆，我们就将业界特指为科技界。

首席数据科学家将为公司设定数据策略，这包括筹建用于收集数据和记录日志的基础架构，确定如何在收集数据的同时保护隐私、哪些数据是面向用户的、如何使用数据做决策，又如何把这些数据反过来应用于产品设计，提升产品质量。他要管理一个由工程师、科学家和分析师组成的团队，还要负责和公司的管理层（如CEO、CTO等）进行沟通。他还负责为创新性的成果申请专利和设定研究目标。

更广泛地看，数据科学家是这样一种人，他懂得如何从数据中抽取信息并且解释数据背后的意义，这需要掌握统计学和机器学习中的工具和方法，还要具备人文主义精神。他要花费大量时间来采集、清理和处理数据，因为数据永远都不会是整齐规范到让人一眼可以读懂的。在这个过程中，他需要坚持不懈，需要统计学和软件工程的技巧，而这些也是理解数据的偏差、调试程序时所必备的技能。

当数据被整理成型后，他需要结合可视化和数据的意义对数据进行探索分析。他会找出模式，构建模型，设计算法——有些是为了了解产品的使用情况和整体质量，有些是为了搭建原型，将在这些原型上经过验证的东西重新揉入产品中，从而提升产品品质。他会设计实验，他是基于数据做出决策这一过程中的关键一环。他要使用明白无误的语言和图形同组内成员、工程师、领导层交流，即使有人对数据不是很敏感，也可以通过他知道这些数据背后的意义。

这就是对数据科学的一个概览，本书将帮助你了解其中大多数内容。接下来，让我们停止空谈，开始进入数据科学的实战阶段！

第 2 章　统计推断、探索性数据分析和数据科学工作流程

在本章，我们首先讨论统计推断和统计学的思考方式，然后我们将目光转向每一位数据科学家都会从事的工作：探索性数据分析。我们还将详细了解研究数据科学的工作流程，在本章的结尾，是我们的“思维实验”环节和一个案例学习。

2.1　大数据时代的统计学思考

> “大数据这个词现在时常被人们随意使用，然而其语义十分模糊。简单地说，这个包罗万象的词条一般有三层含义：首先，它指代一揽子的技术；其次，它有可能引发一场度量数据规模的革命；最后，它为人们未来将会甚或是应该如何制定决策提供了一个新视角，一种新理念。”
>
> —史蒂夫•洛尔（Steve Lohr）
> 《纽约时报》

在你打算成为一名数据科学家时，以下技能是必须首先具备的：统计学、线性代数和一些编程技能。同时你还需要发展以下技能：数据预处理、数据再加工、数据建模、编码、可视化和有效沟通，这些技能往往是相辅相成的。这些内容会在本书中交叉出现，万丈高楼平地起，让我们先从统计推断开始探索数据科学的旅程。

作者希望本书的读者拥有五花八门的背景，你可能是位优秀的软件工程师，有能力搭建数据管道，但对统计学却所知甚少；或者你是一位市场分析专员，一点也不懂如何编写程序；或者你只是一位对数据科学充满好奇的读者，想弄明白数据科学到底是什么。

虽然我们在这里列举了阅读本书需要具备的一些先决条件，但我们不是查户口的，无法到你家去检查你是否修习过有关统计学的课程，或者是否曾经看过有关统计学的书籍。即使你曾经选修过诸如统计学导论之类的入门课程，但正如我们在鸡尾酒会上经常听到的那样，99%的人都十分惧怕统计学，宁愿从来没上过这门课。如此说来，你不大可能从这些课程中真正领略到了统计学的美妙，更不可能深入研究

它。

即使你取得了统计学的博士学位，已经对这一领域有精深的研究，你也可以通过阅读本书回顾一些基础概念，记起什么是统计推断、什么是统计学的思考方式，这总是有所助益的，特别在“大数据”时代这个全新的语境下，很多传统的统计学方法可能都需要重新审视和修订。

2.1.1 统计推断

我们所处的世界异常复杂，充满了随机性和不确定性，同时，这个世界又是一个巨大的数据生产机器。

我们搭地铁或开车去工作，我们的血液在身体里流动，我们购物、收发电子邮件、因浏览网页或查看股价而耽误正事；我们工作、吃饭、和朋友家人聊天；工厂里生产产品……这些行为都会直接或间接产生数据。

试想花一天时间去观察窗外的人流，记录下每一个经过的人；或者聚集起住在你一公里以内的人，问他们在过去的一年中每天收到多少封电子邮件；或者去你附近的社区医院翻阅血液样本，去发现蕴藏在其中的DNA模式，这些行为乍一听觉得有点变态，让人觉得不安，但事实并非如此。我们的生活本身就是不断产生数据的过程。

我们想用多种方式去描述、揭示这些过程，使人们更好地理解数据的产生。作为科学家，我们想更好地了解这个世界。对过程的了解掌握，也是解决问题的一部分。

数据就是现实世界运转留下的痕迹。而这些痕迹会被如何展示出来，则取决于我们采用什么样的数据收集和样本采集方法。假如你是数据科学家，那么作为一个观察者，你要做的事是将具象的世界转化为抽象的数据，这个过程是绝对主观的，而非人们所想的那么中立客观。

将这一过程从数据收集中剥离出来，就能清晰地看到蕴藏其中的随机性和不确定性的两个源头：一是来自过程本身，二是来自数据采集方法。

从某种程度上说，掌握了数据就掌握了世界，或者世界的运作规律，但是这并不代表着，你拿着一个巨大的Excel文件，或者存有数百万条记录的数据库，手指轻轻一划，就能理解世界及其运行规律。

你需要新的点子，将这些采集到的数据进行简化，使它们更易于理

解，能够以一种更简明扼要的方式概述世界运行的规律，能够易于使用数学对其进行建模的数据，这称为统计估计量。

这一套从现实世界到数据，再由数据到现实世界的流程就是统计推断的领域。

更准确地说，统计推断这门学科主要关注如何从随机过程产生的数据中提取信息，它是流程、方法和理论的统一。

2.1.2　总体和样本

让我们先来统一一些术语和概念。

在经典统计学理论中，有总体和样本之分。英语中总体和人口是同一个单词，因此一说这个词，人们就会马上联想到：美国人口总数3亿、全世界人口总数70亿等。但是，在统计推断中，总体并不特指人口，它指的是一组特定的对象或单位，比如推特上发布的消息，照片或者天上的星星等。

如果我们可以度量和提取这些对象的某些特征，就称为对总体的一组观察数据，习惯上，使用N表示对总体的观察次数。

假设总体为去年“巨无霸”公司员工发送的所有电子邮件，则对该总体的一组观察数据可以包括以下内容：发信人的姓名、收信人列表、发信日期、邮件内容、邮件字数、邮件中的句子数、邮件中动词的个数、从邮件发出到获得第一次回复中间的时间等。

接下来就要采集样本。所谓样本，是指在总体中选取的一个子集，用n来表示。研究者记录下样本的观察数据，根据样本特征推断总体的情况。采样的方法多种多样，有些采样方法会存在偏差，使得样本失真，而不能被视为一个缩小版的总体，去推断总体的特征。当这种情况发生时，基于样本分析所推断出来的结论常常是失真甚或完全错误的。

在上述“巨无霸”公司的例子中，可以随机抽取全部员工的1/10，以他们所发的邮件组成样本，或者随机选取一天，以该天内所发邮件的1/10作为样本。这两种抽样方式都是合理的，而且样本的数量也是相等的，但是，如果你据此计算每人发送电子邮件的数量，进而估计出“巨无霸”公司员工的发送邮件分布情况的话，应用两种采样方式，你将会得到完全不同的答案。

这样一个简单的问题，由于采样方式的不同，结果都会失真，那么对于那些复杂的算法或模型，如果你没有把获取数据的方式考虑进来，情况又会怎样？

2.1.3 大数据的总体和样本

在大数据时代，我们有能力记录用户的所有行为，我们难道不就可以观察一切？那么，此时做总体和样本的区分还有意义吗？如果我们已经拥有了所有的电子邮件，干嘛还要采样？

这些疑问直抵问题的核心，对此，我们有如下几个方面需要加以澄清。

采样可以解决一些工程上的挑战
在时下流行的关于大数据的讨论中，企业都主要采用Hadoop等分布式技术去解决海量数据带来的工程及计算问题，但他们却忽略了采样这种手段也同样有效。事实上，在谷歌，软件工程师、数据科学家和统计学家时刻都在用到采样来处理大数据。

使用多少数据取决于你的目标：比如，做分析或推断，你只需要部分的数据就可以了；但当你试图在用户界面上展示其中一个使用者的信息时，你可能需要搜集该使用者的所有数据。

偏差
即使我们拥有了谷歌、Facebook或Twitter的所有数据，基于这些数据所做出的统计推断并不适用于其他不使用这些服务的人群，即使针对使用这些服务的人群，上述统计结论也不能准确说明他们某一天的活动轨迹。

微软研究院的Kate Crawford女士在她的演讲“Hidden Biases of Big Data”中提到，如果仅对飓风桑迪到来前后的推特做分析，可能会得出这样的结论：人们在飓风来临前在购物，飓风过后在聚会。然而，这些推特大部分来自纽约人。首先，他们是推特的重度用户，而这时沿海的新泽西人正在担心他们的房子会不会被飓风吹倒，他们哪里还有时间和心情去发推特呢？

换句话说，如果你使用推特数据来分析桑迪飓风的影响，得出的结论可能会是：这场飓风危害不大。事实上，你得出的结论只能说明飓风对推特用户的影响。他们受桑迪飓风的影响很小，还有时间来发推特，但他们无法代表一般意义上的美国大众。

同样在这个例子中，如果你不了解相关的语境或者对桑迪飓风一无所知，你就不可能对数据做出合理的解释。

采样

让我们再来看看总体和样本在各种语境下的含义。

在统计学中，我们通常使用一种基础的数学方法来建模，描述总体和样本的关系。针对背后可能存在的规律、数学结构以及生成数据的过程，我们会做出一些简单假设。每一次研究，我们对其中一种数据生成过程中所采集到的数据进行观察，这组数据就是所谓的样本。

以“巨无霸”公司的邮件为例，如果我们随机抽取阅读一些邮件，就会产生一个样本。但如果我们再次抽取，又会产生一组完全不同的样本。

由于采样过程不同所带来的不确定性有一个学名：取样分布。就像2010年上映的、由莱昂纳多·迪卡普里奥主演的《盗梦空间》一样，这是一个梦中梦。因此，也可以将“巨无霸”公司的电子邮件想象成一个样本，而不是总体。

这些电子邮件是一个更大的超级总体的样本（此刻，我们有点哲学家附体），如果抽样时是用扔硬币来决定的话，硬币若多翻转一次，得出的样本就会完全不同。

在这里，我们使用一组电子邮件作为样本，来推断其中一个数据产生的过程，比如说：“巨无霸”公司员工的电子邮件书写习惯。

新的数据类型

过去，所谓数据是指数字和一些分类变量，但今非昔比。在大数据时代，一个优秀的数据科学家需要多才多艺，要处理的数据种类比过去要多得多，如下所示。

- 传统数据：数字、分类变量和二进制变量。
- 文字：电子邮件、推特、《纽约时报》上的文章（详见第4章和第7章）。
- 记录：用户数据、带有时间戳的事件记录和JSON格式的日志文件。
- 地理位置信息数据：将在本章使用纽约市的房屋数据为例做以简单说明。
- 网络（详见第10章）。
- 传感器数据（不在本书讨论范围内）。

- 图片（不在本书讨论范围内）。

这些新的数据类型要求我们在做采样时需更谨慎。

以Facebook用户产生的实时数据流为例，从带时间戳的日志上，可以抓取用户一周内的活动数据，在此基础上进行分析，得出的结论适用于下周或者下一年吗？

在复杂的网络中你又如何采样，使得样本可以反映总体的复杂性？

这些问题都是统计学和计算机科学领域内的开放性研究问题，我们本来就身处科技的前沿。在实际中，数据科学家尽其所能去解决这些问题，在他们的工作中，经常发明出一些创新性的方法。

> 术语：大数据
>
> 我们已经多次提到了“大数据”，但是一直没有好好定义它，只是在上一章中围绕它提出了一些问题，我们也觉得不好意思，那就让我们先来看看什么是大数据。
>
> 大数据的大是相对的。人为地为大数据限定一个阈值，比如1PB，是没有意义的，这太绝对了。只有当数据的规模大到对现有技术（比如内存、外存、复杂程度、处理速度等）构成挑战时，才配称为“大”。因此，大数据的大是一个相对概念，大数据放在20世纪70年代和现在的意义是完全不同的。
>
> 当用一台机器无法处理时，就可以称为“大数据”。不同的人、不同的公司，拥有的计算资源是有差别的，对于数据科学家来说，如果数据大到一台机器处理不了，就可以称其为“大数据”，因为她不得不学习使用一些全新的工具和方法去解决这一问题。
>
> “大数据”是一种文化现象。它描述了数据在人类生活中所占的比重，随着科技的发展，数据所占的比重越来越大。
>
> 大数据中的**4V**原则。这4V是指容量（Volume）、种类（Variety）、速度（Velocity）和价值（Value）。很多人借此来描述大数据的特征，你也可以从中学习借鉴。

2.1.4 大数据意味着大胆的假设

还记得在第1章，我们提到了库克耶和迈尔-舍恩伯格的文章“The Rise of Big Data”，在文章中，他们提出大数据的革命由以下三方面构成：

- 采集和使用大量的数据，而不是小样本；
- 接受数据中存在杂乱噪声；
- 重视结论，放弃探究产生结果的原因。

他们将这三方面说得冠冕堂皇，他们宣称，数据是如此巨大，没有必要去寻找原因。也不用担心采样出错，因为所有的数据都在这，它们记录了一切事实。之所以这样说，是因为他们声称找到了处理大数据的新方式，那就是让“N ＝ 全部”。

***N*能代表全部吗？**

答案是N永远不能代表全部。我们经常忽略那些我们最应该关心的事实。

以InfoWorld上的一篇帖子为例，互联网监控永远也不可能奏效，我们最想抓住的那些罪犯，恰恰是非常聪明、精通技术的，他们永远领先一步，永远也不会被人抓住。

那篇文章举了一个选举之夜投票的例子，这个例子中本身就存在矛盾之处：即使我们能把所有离开投票站的人都纳入统计，我们也还是遗漏了那些从一开始就不打算投票的人，他们那天晚上压根儿就没来投票站。而这些人或许才是我们需要关注的人群，和他们交谈才能了解我们国家现有投票制度存在的问题。

事实上，我们认为“N ＝ 全部”这个假设是大数据时代人们面临的最大问题。首先，这种假设天然地将一大部分人排除在外，他们可能是因没有时间、精力、渠道去参加那些非正式或者未经宣布的选举投票。

在统计选票时，我们忽略了那些同时打两份工的人，还有那些把时间都花在等公交车上的人，他们因为各种原因没有投票。对你来说，这或许只意味着Netflix的推荐引擎推荐给你的电影不符合你的口味，因为在Netflix上愿意费心去为电影打分的大多是年轻人，他们的口味可能和你的不一样，这些打分行为使得推荐引擎更贴近他们的喜好。但是，上述例子中提出的基本观点在现实生活中很可能会埋下许多潜在隐患。

数据是不客观的

若说“N = 全部”这一假设可以成立，则意味着要承认数据是客观的。但是，相信数据是客观的，或者“让数据说话”，这些观点是错误的，当然，对于那些持完全相反观点的人也应该小心提防。

最近，我们在《纽约时报》上发表了一篇关于运用大数据方法来招聘人员的文章（http://nyti.ms/18OEq6j），正是这篇文章使我们意识到上述观点的可怕。文章中引述了一位数据科学家的说法：“让我们把所有东西都放进来，让数据自己说话。”

通读全文后，你会认识到，运用大数据运算法则是为了寻找到有潜质的“璞玉”式人才。这种做法在招聘中值得一试，但是有些事你要深思熟虑。

假设你面前有两位资历相当的求职者，一位是男性，一位是女性。阅读他们的简历你发现，相比之下，这位女性求职者跳槽次数更多，获得提拔的机会较少，而且对以前工作过的地方有更多的负面评价。

当再有这样的情况出现时，你若通过模型做决策，可能会雇用男性而不是女性求职者。你的模型是不会把有些公司歧视女性的情况考虑进去的。

换句话说，忽视因果关系是大数据法则的一种缺陷，而不是特征。忽视因果关系的模型无助于解决现存问题，而只会增加更多问题（在第11章将会予以详细说明）。数据也不会自己说话，它只能够以一种量化的、无力的方式去描述、再现我们身边的社会事件。

n = 1

与“N = 全部”这一极端观点相反的是另一个极端：n = 1，意思是说样本的总数为1。过去，说样本空间的大小为1是很荒谬的，没人会通过观察一个个体，就得出对总体的推断。别担心，这种观点现在仍然是荒谬的。不过，在大数据时代，n = 1有了新的含义，对于一个人，我们可以记录所有关于他的信息，我们可以对所有举动进行采样（比如他打过的电话、他敲击键盘的记录），并对这些行为进行统计推断。这是一种用户级别的建模。

2.1.5 建模

下一章将会讲述对于采集到的数据如何进行建模，本章，我们先讨论什么叫建模。

Rachel曾经给一位朋友打电话，讨论建模交流会的事，几分钟后她意识到，“model”这个单词对他俩来说完全是不同的含义。对方理解成了数据模型（data model），一种存储数据的结构，这属于数据库管理员的研究领域。而Rachel的意思是统计学中的模型，这也是本书大部分时间要讨论的内容。更可笑的是，最近Andrew Gelman的一篇关于建模的博客被时尚圈的人士发到了推特上，他们可能理解成了模特。

即使你已经使用统计模型或数学模型这两个术语很多年了，但当你向周围人谈起模型时，你和他们都明确知道这个词的含义吗？什么使一个模型成其为模型？当我们问起这些基本问题时，还有一个问题也很重要，统计模型和机器学习算法之间有什么不同？

在深入这些问题之前，让我们先来看看Chris Anderson 2008年在《连线》杂志上发表的文章“The End of Theory: The Data Deluge Makes the Scientific Method Obsolete”，这篇引人争议的文章为我们的讨论增加了更多素材。值得一提的是，Chris Anderson那时候是《连线》杂志的主编。

Anderson认为数据即信息，并且宣称不需要模型，了解相关性就够了。以海量数据为例，“谷歌根本没有必要使用模型”。

是这样吗？我不相信，读完此书，我觉得你们也不会相信。这种观点类似库克耶和迈尔-舍恩伯格在他们的文章中提出的“N = 全部”，我们刚刚在前面讨论过。现在，你也许已经可以感受到环绕在我们周围的深深的疑惑了吧。

然而我们需要感谢媒体，是他们让公众了解了这些问题，可是，当这些意见领袖们并不是专职从事数据科学工作的人员时，对他们的说法就得打一个问号了。仔细想想你是否同意Anderson的观点，哪些部分你认为是对的，哪些地方你认为是错的，或者你需要获取更多信息才能形成自己的观点。

鉴于公众对数据科学和模型的认知都来自主流媒体这样业余的描述，作为数据科学家，我们应该义不容辞地发出自己的声音，贡献出自己的真知灼见。

在这个大背景之下，当我们谈起模型时，其含义究竟是什么？数据科

学家是如何使用它们的？让我们直接进入下一节，来回答这些问题。

什么是模型？

人类试图用各种方式去描述他们所处的世界。建筑学家用蓝图和三维立体模型来捕捉建筑的属性；分子生物学家用连接氨基酸的三维图像描述蛋白质结构；统计学家和数据科学家则用函数表示产生数据的过程中存在的不确定性和随机性，并以此来形容数据本身的样貌和结构。

模型就好像一个特殊的镜片，我们透过这个镜片去观察和了解现实世界的本质，这个“镜片”可能是建筑学、生物学或数学模型。

模型是人工设计的，用于将无关紧要的细节排除或抽象化。在进行模型分析时，研究者必须关注这些被省略的细节。

以蛋白质为例，一种本身带有侧链的蛋白质骨架却不受规范电子运行轨迹的量子力学理论的约束，最终决定了蛋白质的构架和行为。以统计模型为例，建模时我们可能错误地排除了一些关键变量，而使用了一些与问题无关的变量，或者采用的数学结构偏离了问题本身。

统计建模

在引入数据和开始编写代码前，对于建模的流程有一个大概的了解是有益的。先干什么？谁受谁的影响？什么是因，什么是果？检验结果如何？这些都是我们应该思考的问题。

人们使用的方法各有不同，有些人喜欢用数学去描述这种关系。通用的数学公式里面必须包括参数，但是参数的值是未知的。

按照惯例，数学公式里一般使用希腊字母表示参数，拉丁字母表示数据。比如你有两列数据：x和y，你认为二者之间是线性关系，用公式表达如下：$y = \beta_0 + \beta_1 x$，此时你还不知道β_0和β_1的具体数值，因此，它们就是参数。

还有些人则喜欢画图。他们先画一张数据流的图，很可能带有箭头，用来描述事物之间是怎么相互影响的，或者在一段时间内发生了些什么。在选择公式表达这种关系之前，这种关系图可以给他们一个大概的描述。

如何构建模型

你怎么知道什么数据该用什么模型？这一半是艺术，一半是科学。这个问题正是打开数据科学大门的钥匙，可惜的是，本书中就这个问题能够给出的指引非常有限。只能说模型的选择是建模过程中的一环，你需要对底层结构做出大量假设，应该有一个标准来规范如何选择模型和解释这样选择的理由。但是我们还没有统一的规范，所以只能摸着石头过河，希望经过深思熟虑，能制定这样一套规范。

必须承认，我们也不知道从哪儿开始，如果知道的话，我们已经知道了生命的意义。但是，我们会尽力，尽力在书中向你展示我们在面对这样的问题时要怎么做。

也许探索性数据分析（EDA）是一个好的开始，我们将在下面一节介绍它。它牵扯到绘制图形和从数据集中获取直观的感觉。与试错、反复实验一样，探索性数据分析对问题的解决大有帮助。

实话实说，除非你做过很多次，否则这一过程在你眼里依然是那么神秘。最好是从易到难，先做看起来最傻的事，事后看，或许没有你想象得那么傻。

举例来说，你可以（或者说是应该）先绘制直方图或散点图以对数据产生一个直观的感受，然后试着写点什么，哪怕一开始是错误的（很可能一开始的结论是错误的，没关系）。

比如写出来的是一个线性方程（详见下一章），当你把它写下来，就会强迫自己去思考：这个方程有意义吗？如果没有，为什么没有？那怎样的方程对这个数据集是有意义的？你从最简单的方式开始，逐渐增加复杂度，做出假设并把你的假设写下来。如果你觉得完整的陈述有帮助，就把句子写完整，比如："我假设将用户自然分成五组，因为销售代表谈起用户的时候，她把用户分成了五类。"然后试着用方程式和代码来表达这一陈述。

记着，从简单处着手永远是个好办法，建模时在简单和准确之间有一个权衡。简单的模型易于理解，很多时候，原始简单的模型帮你完成了90%的任务，而且构建该模型只需要几个小时，采用复杂的模型或许会花上几个月，而且只将这个数值提到了92%。

在本书中，你将从构建模型开始，它们将组成你的"兵工厂"。在构建模型时会用到很多模块，其中一种就是概率分布。

概率分布

概率分布是统计模型的基础。在讲述线性回归和朴素贝叶斯时，你就会知道我们这么说的原因。概率论是一门需要花费几学期去教授的课程，将这样庞杂的内容压缩并在一节中讲述，对我们来说实在是个巨大的挑战。

回到还没有发明计算机的年代，科学家观察到了现实生活中的一些现象，经过测量后发现，一些固定的数学模式在重复出现。其中的经典案例莫过于人的身高服从正态分布，正态分布是一种钟形曲线，也叫高斯分布，以数学家高斯的名字命名。

还有其他一些经常出现的曲线，也分别以各自的发现者的名字命名（比如泊松分布、韦伯分布等），另外一些曲线，如伽玛分布、指数分布，则是以描述它们的数学方程式命名。

自然状态下产生的数据，可以用数学函数来描述。通过设定函数中的参数，可使函数曲线接近于实际数据的分布形态。而这些参数可在对数据进行估计的基础上得出。

不是所有过程产生的数据都服从某种已知的分布，但很多都服从。我们可以应用这些分布函数作为模块，来构建最终的模型。本书并不打算深入探讨每种分布的细节，但我们提供了图2-1用来展示这些常用的分布，事实上有无穷多种分布，列在这里的这些只是因为有人观察了它们很久，觉得有必要给予命名而已。

概率分布可以理解为对于可能结果的子集指定一个概率，概率分布用与其对应的函数来表示。比如正态分布的函数为：

$$N(x|\mu,\sigma) \sim \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

参数μ是平均值或中位数，决定了该分布的位置（正态分布是一种对称的分布）。参数σ决定了分布的幅度。这是一般意义上的方程式，在真实世界的各种特定现象中，这些参数的值是固定的，我们可以通过对数据的估计得到这些参数。

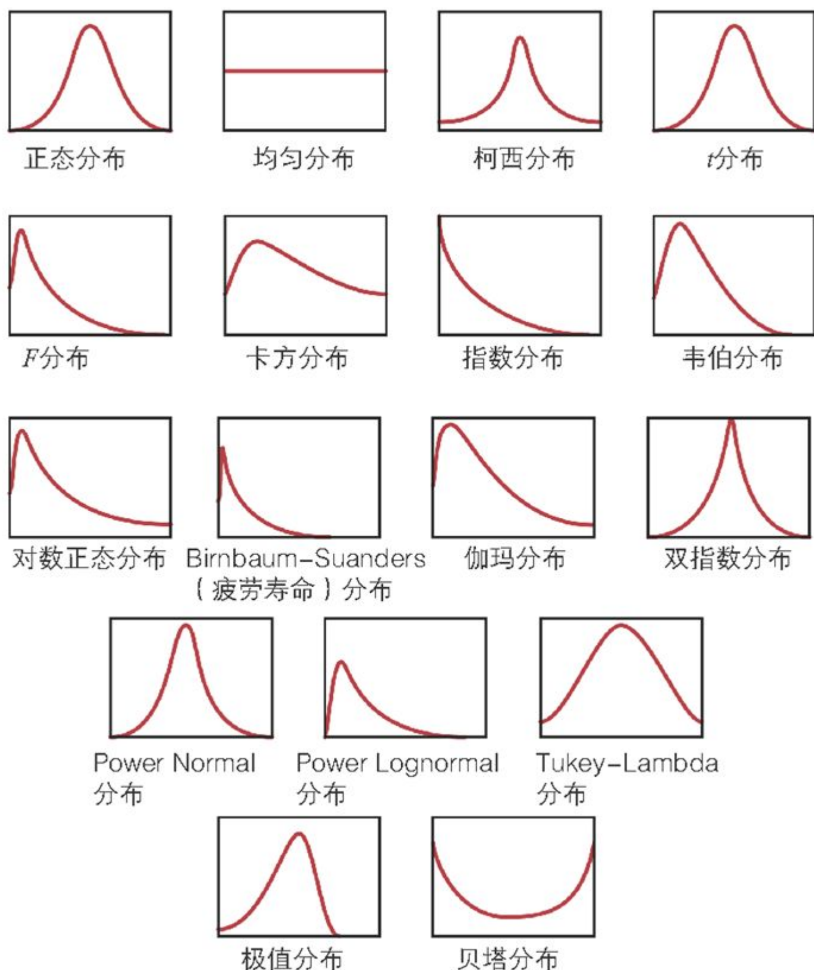


图2-1 一组连续的密度函数（也叫概率分布）

假设随机变量x的概率分布为$p(x)$，该函数将x映射为一个实数，要使其成为概率密度函数，需要对其做以下限定：使用积分求曲线覆盖下的面积，则其值必须为1，这样才能称其为概率。

比如，设x为距离下趟公交车到站的时间（以秒为单位），则x是一个随机变量，因为下趟公交车的到站时间是不定的。

假设我们已知（为了便于讨论）这个等待时间的概率密度函数为$p(x)=2e^{-2x}$，如果我们想知道下一趟车在等候12～13分钟后来的可能

性，则对于12至13之间该概率分布曲线下的区域使用积分$\int_{12}^{13} 2e^{-2x}$求面积即可。

怎么知道该使用哪种概率分布？有两种方法：首先，可以做实验。我们可以随机到达公交车站，测量等候下一趟公交车需要的时间，重复该实验多次。然后将测量得到的数据绘制成散点图，看看与哪种概率分布曲线吻合。或者基于我们对“等待时间”是一种普遍的自然现象这一事实的了解，马上会想到采用指数分布$p(x) = \lambda e^{-\lambda x}$去描述，指数分布就是专门发明用来描述自然界这种普遍现象的。

使用单变量函数可以描述一个随机变量的分布，描述多个随机变量则需要使用多变量函数，这称作联合分布。以两个随机变量为例，使用函数$p(x, y)$表示概率分布，输入为平面上的点，输出为一个非负数。为了确保其是一个概率分布函数，在整个平面求二重积分，其值为1。

还有一种分布叫条件分布$p(x|y)$，其含义是当y给定时x的概率密度函数。

处理数据时，条件意味着一个子集。比如，假设我们有Amazon.com网站的一组用户数据，该数据列出了每个用户上月在该网站的消费金额，不论性别，也不管在将第一件商品加入购物车前浏览过多少商品。

设随机变量X表示消费金额，则可以用$p(X)$表示用户的消费金额分布。

从所有用户中选取一个子集，该子集的用户在购买任何商品前至少浏览过5件商品，让我们看看这些用户上月消费金额的概率分布。设随机变量Y表示在购买第一件商品前浏览的商品数量，则$p(X|Y > 5)$表示条件分布。条件分布和一般的分布拥有一样的性质：求积分的结果为1，而且永远不会为负数。

当我们观察这些数据点时，如(x_1, y_1) ,(x_2, y_2), . . ., (xn, yn)，我们是在观察随机变量的实例。当我们有一个n行k列的数据时，我们是在观察k个随机变量组成的联合分布的n个实例。

如果读者还想了解更多关于概率分布的知识，请参考Sheldon Ross所著的*A First Course in Probability*一书（Pearson）。

拟合模型

拟合模型是指用观察数据估计模型参数的过程。以数据为依据，近似模拟现实中产生数据的数学过程。拟合模型经常要引入各种优化方法和算法，例如最大似然估计等，来确定参数。

事实上，当你估计参数时，参数就成了估计量，他们本身就是数据的函数。当模型拟合成功，就能以数学函数的形式表达，比如$y = 7.2 + 4.5x$，其含义是，基于你对数据间存在线性关系的假设，该函数准确表达了两个变量之间的这种关系。

拟合模型的过程就是开始编写代码的过程：代码将会读入数据，将写在纸上的公式翻译成代码，然后使用R或者Python中内建的优化方法，根据数据，求出尽可能精确的参数值。

等你变得越来越老练，或者这本身就是你的强项时，你可能会去研究这些优化方法。首先得知道这些优化方法的存在，然后弄明白它们是怎么工作的，但是你不必亲自去编写代码实现这些方法，R和Python已经帮你实现好了，直接调用就行。

过拟合

在本书中，不时会提醒你小心过拟合，甚至这会成为你的梦魇。过拟合是指使用数据去估计模型的参数时，得到的模型并不能模拟现实情况，在样本以外的数据上效果不好。

在试图用该模型去预测另一组数据（该组数据未用来拟合模型）的标签时，你可能会发现结果不尽如人意，以准确度去衡量，这并不是一个很好的模型。

2.2　探索性数据分析

> “面对那些我们坚信存在或不存在的事物时，“探索性数据分析”代表了一种态度，一种方法手段的灵活性，更代表了人们寻求真相的强烈愿望。”
>
> — John Tukey

前面我们提到过探索性数据分析是建模的第一步。探索性数据分析经常是标准统计学入门教材的第1章（第1章的意思是，这是最简单、最初级的内容），然后全书就再也不会提及它，它被遗忘了。

探索性数据分析经常表现为画一些直方图或者茎叶图，小学五年级都

开始教这些知识了，因此探索性数据分析看起来只是小菜一碟，不是吗？这也就难怪没人把它当回事了。

然而探索性数据分析是数据科学中的重要一环，同时代表了来自贝尔实验室的一批统计学家在从事数据科学工作时所采用的方法和观点。

John Tukey是贝尔实验室的数学家，他开发出有别于验证性数据分析的探索性数据分析，如上节所述，验证性数据分析偏重于模型和假设。在探索性数据分析中，没有假设，也没有模型。这里的“探索性”是指你对待解问题的理解会随着研究的深入不断变化的。

> 回顾贝尔实验室的历史
>
> 贝尔实验室始于20世纪20年代，在物理学、计算机科学、统计学和数学上做出了很多重大发明和创新，程序设计语言C＋＋也诞生于贝尔实验室，该实验室还培养出很多诺贝尔奖获得者。这里有一支高产且成功的统计学小组，其中数学家John Tukey尤其著名，他解决了很多统计学问题。他被誉为探索性数据分析和R语言之父（R的前身是贝尔实验室的S语言，R是S语言的一个开源版本），他同时对高维数据的可视化也很感兴趣。
>
> 我们认为贝尔实验室是数据科学的诞生之地，因为那里有海量的复杂数据，还有各学科之间的相互合作。和现在的谷歌一样，那里过去是计算机科学家和统计学家的虚拟游乐场。
>
> 早在2001年，Bill Cleveland在其文章“Data Science: An Action Plan for expanding the technical areas of the field of statistics”中就描述了多学科研究、模型、处理数据的方法（也就是传统的应用统计学）、和数据相关的计算（硬件、软件、算法、编码）、教学法、工具评价（现在依然是前沿科技）以及理论（数据背后的数学）。
>
> 读者可以阅读Jon Gertner所著的*The Idea Factory*（Penguin Books）一书了解更多关于贝尔实验室的故事。

探索性数据分析的基本工具是图、表和汇总统计量。一般来说，探索

性数据分析是一种系统性分析数据的方法，它展示了所有变量的分布情况（利用盒形图）、时间序列数据和变换变量，利用散点矩阵图展示了变量两两之间的关系，并且得到了所有的汇总统计量。换句话说，就是要计算均值、最小值、最大值、上下四分位数和确定异常值。

探索性数据分析不仅是一组工具，更是一种思维方式：要怎么看待和数据之间的关系。你想理解数据，了解数据的形状，获得对数据的直观感受，想将数据和你对产生数据的过程的理解关联起来。探索性数据分析是你和数据之间的桥梁，它不向任何人证明什么。

2.2.1 探索性数据分析的哲学

> “与其担心如何说服别人，不如先了解到底发生了什么。”
>
> — Andrew Gelman

在谷歌期间，Rachel有幸与前贝尔实验室的两位统计学家，Daryl Pregibon和Diane Lambert共事，他们都是应用统计学领域的专家。正是从他们身上，Rachel学会了将探索性数据分析作为她的最佳实践之一。

是的，即使面对谷歌级别的大体量的数据，他们依然进行探索性数据分析。在互联网企业中，基于和处理小数据同样的原因，探索性数据分析经常被用到，在处理日志数据时，有更多的理由使用探索性数据分析。

使用探索性数据分析有很多重要的原因。包括获取对数据的直觉、比较变量的分布、对数据进行检查（确保数据的规模在你预期范围内，数据的格式是你想要的等）、发现数据中的缺失值和异常值、对数据进行总结。

对于在日志中生成的数据，探索性数据分析可以用于调试记录日志的流程。比如，你通过统计日志数据发现的一些“模式”，很可能其实是由于日志记录流程中出错造成的，因此这些错误亟待修复。如果你怕麻烦从不去调试，你可能会一直认为这些模式是真实存在的。和我们工作过的工程师总是非常感谢我们在这方面提供的帮助。

最后，探索性数据分析确保了产品的性能符合预期。

在探索性数据分析中会引入许多图形，但是我们有必要在这里对探索性分析和数据可视化加以区分。探索性数据分析是数据分析的开端，而数据可视化（将会在第9章介绍）是在数据分析的最后一个环节，用于呈现数据分析的结论。在探索性数据分析中，图形只是帮助你理解数据。

在探索性数据分析中，可以根据对数据的理解优化算法。比如，你正在开发一种排名算法，该算法对你推荐给用户的内容进行排名。为此，你可能需要定义什么是“流行度”。

在决定以何种方式量化“流行度”之前（可行的量化方式有最高的点击率、最多的回复率、大于某一阈值的回复量或者众多指标的加权平均值），你需要先了解数据的运作表现，而做这件事最好方式就是观察你的数据，亲自去实践。

根据数据绘图，并进行比较，这些将会收到意想不到的效果。相比于拿到数据集后、不管三七二十一就运行一个回归模型，这种方法效果要好得多。你之所以选择回归模型，只是因为你知道它怎么用。对分析师和数据科学家来说，在处理数据时，若没有将探索性数据分析视为重要一环纳入到整个研究过程中，这对研究结果极为不利。给自己个机会，把探索性数据分析作为你的数据分析工作流程中的一部分吧！

这里有一些引用文献，帮助你了解探索性数据分析这一最佳实践和其历史背景：

1. *Exploratory Data Analysis*，John Tukey著（Pearson）

2. *The Visual Display of Quantitative Information*，Edward Tufte著（Graphics Press）

3. *The Elements of Graphing Data*，William S. Cleveland著（Hobart Press）

4. *Statistical Graphics for Visualizing Multivariate Data*，William G. Jacoby著（Sage）

5. “Exploratory Data Analysis for Complex Models”，Andrew Gelman（American Statistical Association）

6. John, T. (1962). The Future of Data Analysis. *Annals of Mathematical Statistics*, 33(1), 1-67.

7. “Data Analysis, Exploratory”，作者为David Brillinger，*International Encyclopedia of Political Science (Sage)*（8页摘录）

2.2.2　练习：探索性数据分析

有如下31个数据文件：nyt1.csv、nyt2.csv……nyt31.csv，可以从https://github.com/oreillymedia/doing_data_science。每一个数据文件记录了《纽约时报》五月份每天出现在主页上的广告和广告的点击次数，当然，这组数据是我们伪造的。数据的每一行代表了一个用户，数据共有5列，分别为：年龄、性别（0 = 女性，1 = 男性)、广告显示次数、点击次数、是否登录。

你将使用R处理这些数据，R是一种编程语言，设计用来专门做数据分析，使用起来很直观。如果你还没安装过R，请至其官方网站http://www.r-project.org/下载。安装完成后，使用下述命令加载一个文件：

```
data1 <- read.csv(url("http://stat.columbia.edu/~rachel/datasets/nyt1.csv"
```

数据加载成功后，就可以开始进行探索性数据分析了。

1. 创建一个新变量age_group，按年龄将用户离散化，分为"<18"、"18-24"、"25-34"、"35-44"、"45-54"、"55-64"、"65+"总共7组。

2. 对于每天的记录有以上操作。

- 对这7组用户，分别绘出点击率分布图(点击率=点击次数/广告显示次数)。
- 定义一个新变量，基于用户的点击行为将用户分类。
- 探索性地分析这些数据，从图形和数量两方面比较各用户组之间的差异（比如小于18岁的男性和小于18岁的女性，或者已登录用户和未登录用户等）。
- 创建用于描述数据的各种统计量、矩阵。可能的度量项目有点击率、分位数、平均值、中位数、方差、最大值等，可以在每个用户组中单独计算这些统计量。要有所取舍，想一想随着时间的推移，哪些才是重要的、值得去记录的，这样可以压缩数据，同时不失准确地记录用户行为。

3. 现在延长你所分析的时间，将数日内的矩阵和分布情况用图形表示出来。

4. 描述并解释你发现的模式。

示例代码

这里我们给出本练习的一个解决方案的前半部分代码。我们不可能在这一本书中同时教授数据科学和如何编写程序，学习使用一种新语言去写程序需要不断去尝试，你可能会遇到错误，这时去谷歌或stackoverflow网站上搜索是个不错的主意。

当你试图在R中绘图或者建模是，或许其他人已经试过了，与其一个人在那苦思冥想，不如上网看看1。但是在你努力凭自己的能力解决这个问题之前，我们不建议马上看我们提供的答案：

> 1编注：O'Reilly网站上也有很多参考书可供阅读。

```
# Author: Maura Fitzgerald
data1 <- read.csv(url("http://stat.columbia.edu/~rachel/datasets/nyt1.csv"

# 分类
head(data1)
data1$agecat <-cut(data1$Age,c(-Inf,0,18,24,34,44,54,64,Inf))

# 预览
summary(data1)

# 分组
install.packages("doBy")
library("doBy")
siterange <- function(x){c(length(x), min(x), mean(x), max(x))}
summaryBy(Age~agecat, data =data1, FUN=siterange)

# 登录的用户才有性别和年龄
summaryBy(Gender+Signed_In+Impressions+Clicks~agecat,data =data1)

#绘图
install.packages("ggplot2")
library(ggplot2)
ggplot(data1, aes(x=Impressions, fill=agecat))+geom_histogram(binwidth=1)
ggplot(data1, aes(x=agecat, y=Impressions, fill=agecat))+geom_boxplot()

# 根据转化率创建点击，如果没有印象，
# 不必在乎其是否点击，如果没有印象的点击出现，
# 证明我对数据的假设是错误的
data1$hasimps <-cut(data1$Impressions,c(-Inf,0,Inf))
summaryBy(Clicks~hasimps, data =data1, FUN=siterange)
ggplot(subset(data1, Impressions>0), aes(x=Clicks/Impressions,
colour=agecat)) + geom_density()
ggplot(subset(data1, Clicks>0), aes(x=Clicks/Impressions,
```

```
colour=agecat)) + geom_density()
ggplot(subset(data1, Clicks>0), aes(x=agecat, y=Clicks,fill=agecat)) + geo
ggplot(subset(data1, Clicks>0), aes(x=Clicks, colour=agecat))
+ geom_density()

# 分组
data1$scode[data1$Impressions==0] <- "NoImps"
data1$scode[data1$Impressions >0] <- "Imps"
data1$scode[data1$Clicks >0] <- "Clicks"

# 将列转换成一个因素
data1$scode <- factor(data1$scode)
head(data1)

#查看水平
clen <- function(x){c(length(x))}
etable<-summaryBy(Impressions~scode+Gender+agecat,
data = data1, FUN=clen)
```

以下为做该练习其他部分时的提示：

不要将所有数据一次性读入内存。当你某天终于将代码写好时，一次只加载一个数据文件，处理它，输出相关的矩阵和变量，将结果存入一个数据框。在加载下一个文件时，记得移除上一个文件。之所以这样做，是为了让你思考在多个机器之间共享数据时该如何处理。

关于编写代码的建议

在2013年5月发表的一篇文章“How to be a Woman Programmer”中，Ellen Ullman相当好地描述了成为一个程序员需要的素质（让我们暂且不去关注和女性特别相关的部分）：

学习编程的首要条件是对编程本身的热爱，对于探索横亘于人脑和机器之间的神秘空间、如何使机器满足人类的需求抱有强烈的探索需求。

第二个条件是允许失败。编程是一门设计算法的艺术，同时是一门调试错误程序的手艺。用Fortran语言的发明者、著名的计算机科学家约翰•巴克斯的话来说：“你要随时准备着出错，你要有很多方案，努

力工作去发现那些不奏效的方案，不断地这样做，直到找到正确的方案。”

2.3　数据科学的工作流程

结合以上内容，让我们来看看如何定义数据科学的工作流程。你见的数据科学工作者越多，你就越会发现他们的工作流程符合图2-2的描述。贯穿全书，我们会使用各种方法介绍该过程的各个环节和案例。

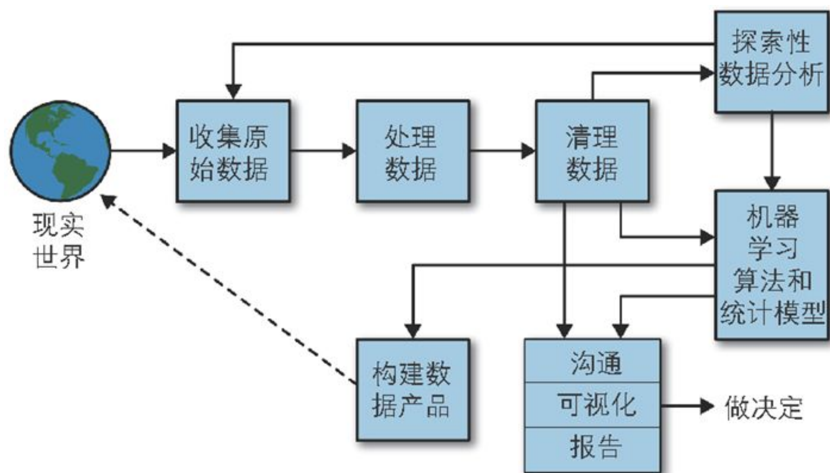


图**2-2**：数据科学的工作流程

首先，我们生活在这个世界中。在这个世界上，有很多人在从事各种各样的活动。有些人在使用Google+，另外一些人则在奥运会上一较高下；有些人在制造、发送垃圾邮件，有些人则在医院里抽血。假设我们拥有其中某项活动的数据。

具体来说，以原始数据为起点，诸如日志、奥运会纪录、安然公司员工的电子邮件、遗传物质记录（需要注意的是，在我们拿到这些原始数据时，这项活动中某些方面的信息已经缺失了）。我们需要处理这些原始数据，使得其便于分析。因此我们创建出管道对数据进行再加工：联合、拼凑、清理，随便你叫它们什么好了，就是要对数据进行再加工。我们可以使用Python、shell脚本、R、SQL完成这件任务。

最终得到格式化好的数据，像下面这种由列构成的数据：

姓名 | 事件 | 年份 | 性别 | 时间



在标准的统计学课程中，通常从一份干净有序的数据文件开始，但在现实中，你通常不会有这么好的运气。

在拿到这份干净的数据后，我们应该先做一些探索性数据分析。在这个过程中，我们或许会发现数据并不是那么干净，数据可能含有重复值、缺失值或者荒谬的异常值，有些数据未被记录或被错误地记录。在发现上述现象时，我们不得不回过头采集更多的数据，或者花更多的时间清理数据。

然后，我们使用一些算法，比如k近邻、线性回归、朴素贝叶斯等设计模型。选取何种模型取决于要解决的问题，这可能是一个分类问题、一个预测问题，或者只是一个基本的描述问题。

这时就可以解释、勾勒、报告或者交流得到的结果。可以将结果报告给老板或同事，或者在学术期刊上发表文章，或者走出去参加一些学术会议，阐述我们的研究成果。

如果我们的目标是开发一款数据产品或其产品原型，例如垃圾邮件分类、搜索排名算法、推荐引擎等。数据科学和统计学的不同之处就体现出来了，数据产品最终会融合到日常生活中，用户会和产品产生交互，交互会产生更多的数据，这样形成一个反馈的循环。

这和天气预报大相径庭，在预测天气时，你的模型对于结果没有任何影响。比如，你预测到下星期会下雨，除非你拥有某种超能力，否则不是你让天下雨的。但是假如你搭建了一个推荐系统，证明“很多人都喜欢这本书”，那就不一样了，看到这个推荐的人没准觉得大家都喜欢的东西应该不会太差，也喜欢上这本书了，这就形成了反馈。

在做任何分析时，都要将这种反馈考虑在内，以此对模型产生的偏差进行调整。模型不仅预测未来，它还在影响未来。

一个可供用户交互的数据产品和天气预报分别处于数据分析的两个极端，无论你面对何种类型的数据和基于该数据的数据产品，不管是基于统计模型的公共政策、医疗保险还是被广泛报道的大选调查，报道本身或许会左右观众的选票，你都要将模型对你所观察和试图理解的现象的影响考虑在内。

数据科学家在数据科学工作流程中的角色

到目前为止，所有这一切仿佛不需要人工干预，奇迹般地发生了。这里说的“人”，是指那些“数据科学家”。总得有人做出决定：该收集哪些数据？为什么要收集这些数据？她还要提出问题，做出假设，制定解决问题的方案。她就是数据科学家，或者她是我们推崇的数据科学团队。

让我们重新修订以前的流程，至少增加一层，来表明数据科学家需要全程参与到这一流程中来，他们不但需要在流程的较高层次上工作，还需要亲手编写程序，如图2-3所示：

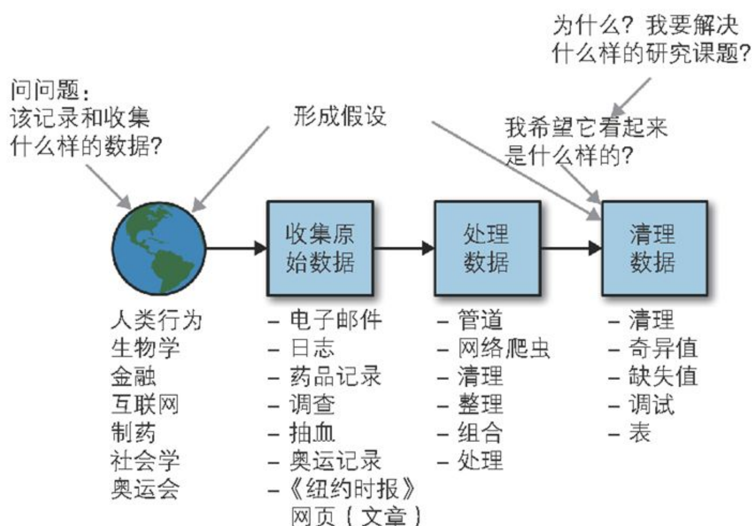


图**2-3** 数据科学家需要参与数据科学工作流程的各个环节

数据科学工作流程和其他科学方法的关系

数据科学工作流程，可以看作是其他科学方法的延伸或变体，它的一般步骤为：

- 提出问题；
- 做一些背景研究；
- 构想假设；
- 做实验验证构想的假设；
- 分析数据并得出结论；
- 把你的结果分享给其他人。

在数据科学工作流程和其他科学方法中，不是每个研究问题都需要按部就班地解决，大多数问题都不用严格走完每一步，几个步骤的组合就可能解决问题。比如，如果你的目标是对数据进行可视化（这本身也可以看成是一个数据产品），很可能你不会使用任何机器学习或统计模型，你只需要想方设法得到干净的数据，做一些探索性数据分析，将结果用图表的形式展示出来即可。

2.4　思维实验：如何模拟混沌

大多数问题一开始都面临一堆脏乱无序的数据，或者问题本身并未得到明确定义，或者问题迫切待解。作为数据科学家的我们，从某种程度上说，肩负着从混沌中恢复秩序的责任。在哥伦比亚大学的课堂上，我们利用课间休息时间讨论了如何来模拟混沌，下面是讨论中出现的一些有意思的观点。

- 洛伦兹水车是一种摩天轮式的精妙装置，它由等间距的叶轮组成，绕轴旋转。每个叶轮下方都有一个小孔，设想一下水流从水车的正上方倾泻而下，从小孔中漏出的水会打到其他叶片上。调整水的流速，会发现叶轮一会儿正转，一会儿反转，呈现一种混沌的状态，请阅读维基百科上的文章了解更多关于洛仑兹水车的内容：http://en.wikipedia.org/wiki/User:Pankajgarg_india/The_lorenzian_waterwheel。

- 很多系统可以演示固有的混沌。Philippe M. Binder和Roderick V. Jensen写过一篇关于利用计算机模拟混沌现象的文章，文章的题目是“Simulating chaotic behavior with finite-state machines”。

- 来自麻省理工、哈佛和塔夫茨大学的研究者发起了一个跨学科的项目，旨在教授“Simulating chaos to teach order”这项技术。他们模拟了发生在乍得和苏丹边境的达尔富尔地区的一起紧急事件，学生在其中扮演无国界医生组织成员、国际医疗队和其他一些人道主义机构。

- Joel Gascoigne也写了一篇关于混沌的文章“Creating order from chaos in a startup”。

讲师笔记

1. 在一个组织中充当数据科学家经常要面对各种混乱，从混乱中恢复秩序是数据科学家的职责。因此，在课堂上我会不断地给我的学生模拟各种混乱的场景。但愿学生们明白这只是出于教学的目的，并非老师无能。

2. 我想通过对“混沌”这个词的不同理解，来阐述词汇的重要性和与人沟通时的困难，人们经常并不知道一个词的确切含义，或者他理解的和你想说的南辕北辙。数据科学家要和领域专家沟通，他们很可能不知道什么是“逻辑回归”，但为了不让自己看起来像个傻瓜，或者觉得这是他们“应该”知道的，总之，他们装作了然于胸，不屑向你提问。若两个人在讨论时并不确知对方口中的术语是什么意思，这样的沟通怎么可能有效？同样的，数据科学家也应该多问领域专家问题，确保自己理解领域专家用到的术语（不管他是一位天体物理学家、社交网络专家还是气象学家）。不知道某些术语并不丢人，不知道还不去问才丢人。你很可能会发现，当通过提问题搞清楚一些术语的含义后，你会对问题本身理解得更透彻。

3. 模拟是数据科学中一项极为有用的技术。为一个模型模拟一些假数据可以更好地理解产生数据的过程，还可以帮助调试程序，这是一项很好的实践。

2.5 案例学习：RealDirect

RealDirect公司的CEO Doug Perlson熟习房地产法，有初创公司和在线广告领域的工作背景。他创立RealDirect公司的目标是利用收集到的房地产数据帮助人们买卖房子。

通常人们每七年就会卖掉自己的房子，这经常要借助于房产经纪人和当前的市场数据。但是房产经纪系统和数据质量本身都存在不少问题，RealDirect正是致力于解决这两方面的问题。

首先说房产经纪人，他们经常是“自由代理人”，他们给自己打工，你可以把他们想象成房产销售顾问。这就意味着他们会极力保护自己手中的数据，那些业绩很好的经纪人通常手中握有大量的数据。但是长远来看，这只不过意味着他们比那些菜鸟多掌握一些数据而已。

为了解决这些问题，RealDirect雇用了一些持有执照的房产经纪人，他们一起工作，共享各自掌握的信息。RealDirect给卖房者提供了接口，给他们一些基于统计数据的卖房建议。RealDirect还利用和用户的交互数据，实时地指导用户该如何进行下一步操作。

这些被雇用的房产经纪人现在成了数据专家，他们学着利用一些数据采集工具收集有用的新信息，或者直接访问那些公众可见的数据。比如，现在你能获取合作式公寓（纽约的一种公寓）的销售数据，这得益于最近的政策变化。

公开的数据有一个缺点：时效性不强。一笔房屋交易完成后，通常要等三个月左右才会被录入公开可查询的数据库中。而RealDirect正致力于向用户提供实时反馈，包括用户何时开始搜索房屋、初始报价是多少、从放盘到成交需要多长时间、用户是如何在网上搜索房屋的……这一系列问题，RealDirect都能为用户提供有效信息。

最终，如果买方和卖方都诚实守信，这些优质的信息对双方都有帮助。

2.5.1　RealDirect是如何赚钱的

首先，它向卖方提供每月大概要花费395美元的订阅服务，用以访问网站提供的销售工具。其次，它允许卖方以较低的价格使用公司的房产经纪人，通常是房屋总价的2%，而市面上的价格一般是2.5%或3%。这时，共享信息的优势就体现出来了，这种更优化的人力资源组织方式允许RealDirect采取更低的定价策略，从而带来更多的生意，利润自然增加了。

RealDirect网站更像一个平台，在这个平台上，买方和卖方可以管理他们的买卖流程。网站实时反映了用户的当前状态：活动、交易达成、交易被拒绝、看房中、正在签合同中等。软件会根据用户的当前状态给出下一步行动的建议。

当然，RealDirect也面临一些挑战。首先，根据纽约的法律，只有登记造册的房屋才能出现在出售目录上，因此RealDirect网站需要用户注册。对于买家来说，这无疑是人为地被设置了一道障碍，但是那些真正想买房子的人，是不会因为这点小麻烦而放弃注册的。此外，那些不需要注册的网站，比如Zillow，并不是RealDirect的竞争对手，因为它们只提供出售的房屋目录，并不提供其他附加服务。Doug指出，使用Pinterest同样需要注册，但依然有无数用户去注册，因此，需要用户注册对RealDirect来说并不是什么问题。

RealDirect的房产经纪人来自各个房产经纪机构，即使这样，RealDirect仍然收到一些来自房产经纪人的来信，他们谴责RealDirect单方面降低价格的行为损害了整个行业的利益。但同时，如果一个房产经纪人因为某房产在RealDirect网站上售卖，而拒绝带领买主去看房的话，这些潜在的买家就会抱怨，他们也在其他地方看见了该房产，凭什么不让看房。因此，传统的房产经纪人别无选择，即使他们不喜欢RealDirect，也不得不和它做生意。换句话说，这些房屋销售的目录是透明的，传统的房产经纪人没有办法不让他们的买家看见这些房子。

Doug谈到了买家购房时考虑的一些关键因素：附近有没有公园、地铁、学校，同片区或同建筑内相似公寓每平米价格的差异。他们想收集这些数据用到RealDirect的服务中。

2.5.2 练一练：RealDirect公司的数据策略

你被任命为RealDirect公司的首席数据科学家，直接汇报给CEO。公司（当然是假想的公司）现在还没有关于如何利用数据的计划，全指望你提出一套数据策略。下面是一些能帮助你开始指定策略的一些方法。

1. 浏览RealDirect网站，站在用户的角度，想一想买方和卖方分别会如何浏览在各页面、网站的各个页面之间是如何组织的。试着去理解现有的业务模型，想一想如何通过分析RealDirect网站用户行为，帮助企业做出决策、改善产品。提出一些你认为可以用数据回答的问题。

- 你会建议工程师为哪些数据记录日志？你期望的数据文件应该拥有何种格式？
- 如何使用数据汇报和管理产品的使用？
- 从数据中得到的信息又如何反馈给产品和网站？

2. 因为现在还没有数据供你分析（通常在初创型公司中，产品这时还处于开发阶段），此时，你应该借助于一些辅助数据来获取对市场的认识。比如，可以去如下网站下载一些数据文件：https://github.com/oreillymedia/doing_data_science。

你可以使用部分或全部的数据文件。

- 第一个挑战：加载和清理数据。然后进行探索性数据分析，找出哪些数据中有异常值和缺失值，你会采取何种方式处理它

们？最后，确保将数据格式转换为你想要的格式，诸如你认为整数类型的值应该确保其类是整型等。

- 当数据整理干净后，继续探索性分析，将数据间的关系用图形展示出来，将数据在(i)空间和(ii)时间两个维度上进行对比。如果你时间充裕，可以试着找找这些数据里面是否蕴含着什么有意思的模式。

3. 将你的发现总结成一份简报汇报给CEO。

4. 作为数据科学家工作的一部分，经常需要向那些不是数据科学家的人发表讲演，因此理想情况下，你需要掌握一些沟通的技巧，可以将你要表达的信息准确传达给对方。你能想到还应该和哪些人进行交流吗？

5. 大多数人不是房地产行业或电子商务的"领域专家"。

- 跨出自己的舒适区，学会在一个不同的环境中采集数据是否给了你一些启示？使得你知道如何在自己的领域内行事。

- 有时候"领域专家"有他们专有的词汇。Doug是否使用了一些他领域内的专有词汇是你所不理解的（"comps""open houses""CPC"）？有时候，如果你不明白一些专家正在使用的词汇，会妨碍你搞清楚问题。养成提问的好习惯，因为迟早你会碰到一些你不明白的事，这需要持之以恒。

6. Doug提到他的公司并不是必须要有一个数据策略。也没有指定数据策略的业界标准。在你做这个练习的过程中，考虑一下是否有一些你想推荐的最佳实践，可以为电子商务或者你自己的领域指定数据策略提供帮助。

示例**R**代码

下述R代码以上节用到的布鲁克林区的房屋销售数据为例，进行了数据清理和探索性数据分析（练习要求使用曼哈顿的数据）。

```
# 作者: Benjamin Reddy

require(gdata)
bk <- read.xls("rollingsales_brooklyn.xls",pattern="BOROUGH")
head(bk)
summary(bk)
```

```
bk$SALE.PRICE.N <- as.numeric(gsub("[^[:digit:]]","",
bk$SALE.PRICE))

count(is.na(bk$SALE.PRICE.N))

names(bk) <- tolower(names(bk))

## 使用正则表达式清理和格式化数据
bk$gross.sqft <- as.numeric(gsub("[^[:digit:]]","",
    bk$gross.square.feet))
bk$land.sqft <- as.numeric(gsub("[^[:digit:]]","",
    bk$land.square.feet))

bk$sale.date <- as.Date(bk$sale.date)
bk$year.built <- as.numeric(as.character(bk$year.built))

## 做一些探索性分析
## 确保销售价格无异常出现
attach(bk)

hist(sale.price.n)
hist(sale.price.n[sale.price.n>0])
hist(gross.sqft[sale.price.n==0])

detach(bk)

## 只保留有价值的订单
bk.sale <- bk[bk$sale.price.n!=0,]

plot(bk.sale$gross.sqft,bk.sale$sale.price.n)
plot(log(bk.sale$gross.sqft),log(bk.sale$sale.price.n))

## 先看1、2和3家庭住宅
bk.homes <- bk.sale[which(grepl("FAMILY",
            bk.sale$building.class.category)),]
plot(log(bk.homes$gross.sqft),log(bk.homes$sale.price.n))

bk.homes[which(bk.homes$sale.price.n<100000),]
     [order(bk.homes[which(bk.homes$sale.price.n<100000),]
    $sale.price.n),]

## 去除那些看起来不像真实订单的奇异值
bk.homes$outliers <- (log(bk.homes$sale.price.n) <=5) + 0
bk.homes <- bk.homes[which(bk.homes$outliers==0),]

plot(log(bk.homes$gross.sqft),log(bk.homes$sale.price.n))
```

第 3 章　算法

上一章探讨了数据科学中模型应用的概况，本章我们将深入学习算法。

算法是完成分析任务所采纳或者遵循的一整套步骤和规则，它是计算机科学中一个基本概念，可视作计算机科学的基石。设计优雅高效的代码、准备和处理数据以至软件工程开发均以算法为基础。

排序、查找、基于图的计算等问题都是算法能够解决的。然而，对于同一个问题，基于效率和计算时间的考虑，可以选出某个相对最优的算法。当算法要解决的数据分析问题涉及海量数据，或者开发面向客户的产品时，基于效率选择最优的算法会变得尤为重要。

高效的算法是准备和处理数据的基础，算法的执行可以是顺序的，也可以是并发的。就数据科学来说，以下三类算法是必须了解的。

1. 数据清理和预处理的算法。比如排序、MapReduce、Pregel。
我们将这类算法称为数据工程，我们将用一章内容专门介绍这类算法，然而，这不是本书的重点。这并不代表你永远不会用到它们，只是作为一类算法，本书并不予以特殊强调。

2. 用于参数估计的最优化算法。比如Stochastic Gradient Descent（随机梯度下降）、 Newton's Method（牛顿法）和 Least Squares（最小二乘法）。在本书中，我们会介绍这些算法。另外值得一提的是，在R软件中这些算法都有相应的实现函数。

3. 机器学习算法。该类算法是本书的重点，也将占据本书的大部篇幅，下面我们就详细介绍该类算法。

3.1　机器学习算法

机器学习算法的应用主要有三个大舞台：预测、分类和聚类。

且慢！在上一章中你不是刚刚说过，是用模型来做这些事情吗？确实！这里似乎有一点混乱，我们先来澄清一些概念。

统计模型出自统计学家之手，机器学习算法则是计算机科学家所开发

的。但某些技术和方法在统计模型和机器学习算法之中都会用到。因此，在本书中这两个词有时可以换着使用。

你会发现本书中的一些算法，比如线性回归，在机器学习和统计学的书中都会出现。讨论这些算法到底来源于哪个学科没有太大的实际意义。

一般来讲，机器学习是人工智能的基础，比如图像识别、语音识别、推荐系统、排名和个性化推荐系统技术——这些都是打造数据产品的基础。然而传统的统计学院里可能不会设置这些课程。因为这些算法的作用往往不是推断数据背后的生成过程（这是统计推断的目的所在），而是尽可能准确地预测和分类。

Rachel在谷歌工作期间，以及参加一些学术会议时，发现机器学习专家和统计学家在处理问题的方式上也有所不同，这种不同反应了各自学科文化上的差异。但作为数据科学家，应该能够在两种思维模式下自由地切换。

这里我们澄清一些基本概念。

- 关于参数的解释
  统计学家认为模型中的参数必须在现实世界中是有意义的。就拿线性回归模型来说，模型中的参数往往都对应着现实世界中的某种行为或者现象，因此模型可以看作是现实世界的某个缩影。而参数在软件工程师或计算机科学家的眼中又是另一番景象，他们主要关心的是如何将算法（包含其中的参数）植入到数据产品中，模型有些可以很复杂，甚至难以解释。有些模型甚至被叫作黑匣子，因为们根本不知道模型内部到底是如何运作的。他们通常不会关注模型参数的意义。即便他们想要关注，也可能是只是为了提升模型的预测能力，而不是为了解释这些参数。

- 置信区间
  在统计学中，置信区间和后验分布用来描述参数估计的不确定性。但是，有些机器学习算法，比如*k*均值算法（k-means）和*k*近邻算法（k-nearest neighbors），就不涉及置信区间和参数估计的不确定性问题。我们稍后会详细介绍这两个算法。

- 显式假设的角色
  统计模型会对数据的生成过程和数据的分布做出一些明确的假设（称作显式假设），统计推断的过程往往是建立在这些假设

的基础之上。而本章稍后将要介绍的非参数检验方法，并不对概率分布做任何显式假设（可能是隐式的）。1

> 1比如说，分布是对称的，或者光滑的。

可以说，统计学家每天都在和“不确定性”打交道，对任何事情他们都不会100%的确信。而软件工程师喜欢打造产品，他们喜欢构建模型以尽可能精确地做出预测，但他们可能并不关心模型本身的不确定性——只要模型效果好就行！像Facebook和谷歌这样的公司，其理念就是打造产品，并随着新数据的涌入不断迭代与更新产品。一个数据科学家要能够在统计学和计算机科学的思维方式之间找到一个平衡点，要取长补短。数据科学家的身上同时流淌着统计学家和计算机科学家的血液，大可不必厚此薄彼。最后让我们引用客座讲师Josh Wills的话对本节做一总结，他的这番话被推特上很多人引用过：

> “数据科学家是软件工程师中最好的统计学家，是统计学家中最好的软件工程师。”
>
> — Josh Wills

3.2　三大基本算法

对于数学科学家来说，商业和现实世界中的很多问题都可以转化为相应的数学模型的形式，最终都可以归类为分类和预测两大问题。学术界和工业界对这两大问题的研究已经比较成熟了，有很多的算法可以直接拿来用。

作为一个数据科学家，在熟练掌握各种算法之后，真正的挑战其实才刚刚开始。你需要根据特定问题和隐含的假设来推敲到底使用哪个算法或者模型。这种判断部分源自经验：当解决的问题足够多时，每当遇到一个新的问题你可能会思考：“这是一个典型的分类问题，模型的输出变量是一个二元变量”，“这应该也是一个分类问题，但奇怪的是输出变量没有做任何标签”。同样都是分类问题，你却知道应该使用不同的算法处理。（第一个问题，你可能会选择逻辑回归或者朴素贝叶斯模型，第二个问题可以采用K均值算法。稍后我们会更详细地介绍这些算法。）

当你还是个学生或初学者时，乍听这些算法可能会有点不知所措，你时常会想“我怎么知道我想解决的这个问题需要使用哪种算法”。这些都是建模的内功，是需要花时间和精力去领悟的。

有一种人拿着锤子，觉得满世界都是钉子。“我会线性回归，碰到的所有问题我都想用线性回归模型去解决。”千万不要这样，作为一个合格的数据科学家，要不断尝试去了解问题的上下文和问题本身的属性，把它们用数学模型的形式表达出来，然后再想想你学习过的算法中哪些可以用来解决该问题，哪些更加适合这个问题。

如果你还是心里没底，找个懂行的人去深入探讨一番也无妨。问问你的同事，参加一个讨论组，或者在你的周围发起一个这样的讨论组。问题之所以成为问题，就在于它的解决方案不是显而易见的。要时刻保持这样虚心的态度，这样当你去解决一个问题时，就会更加审慎。你不必成为一个建模领域的“百事通”，不要说：“显然，这个问题使用带有惩罚项的线性回归模型就能解决。”千万别轻易下结论，即使有时你认为答案很明显，也要多听听旁人的意见。

之所以说起这些，是因为我们总是笃信教科书教给我们的。教科书总是把问题和解决这些问题的方法都摆出来，然后告诉你哪类问题应该用哪种方法。（比如，用体重预测身高应该使用线性回归模型。）这样的教科书学习模式对于刚开始理解和学习线性回归模型是有一定好处的：因为新的知识被吸收是需要一定时间的，需要多加练习。但是，当你熟练掌握了这项技术后，真正的挑战在于你能否从一开始就知道什么情况下应该使用线性回归模型。这里面存在的挑战往往是教科书不会告诉我们的。

本书不会讨论所有的机器学习算法，毕竟本书不是一本有关机器学习的书。关于算法的书市面上已经有很多了，而且有些写得非常好。

话虽如此，我们还是会在本章介绍三种基本的算法，随着本书后续内容的展开，还会介绍其他算法。学习这些算法的目的其实在于培养举一反三的能力：在遇到新的、非教科书式的问题时，我们要起码能看清解决问题的可能方向。

我们会在书中尽量展示数据科学家在面对一个问题时的思考过程，结合问题的上下文，该如何确定该使用哪种算法。我们建议读者在遇到一个数据问题时，习惯性地让自己思考：这个问题的属性是什么？这些属性如何影响到算法的选择？

这里先介绍一些基本的机器学习算法，让我们以线性回归、k近邻（k-NN）和k均值作为开始。就像之前说的，算法的选择与问题的属性相关，要看这些算法是否适合解决该问题。同时也应该从另一个角度审视这些算法，哪些模式是我们仅凭肉眼就可以发现的？当数据变得复杂时，光凭眼睛观察数据中的模式会变得不现实，这时候，就要及时

借助计算机的帮助了。

3.2.1 线性回归模型

线性回归是统计学中最常用的统计方法之一。从根本上来说，当你想表示两个变量间的数学关系时，就可以使用线性回归。当你使用它时，你首先假设输出变量（有时称为响应变量、因变量或标签）和预测变量（有时称为自变量、解释变量或特征）之间存在线性关系。当然这种线性关系也可能存在于一个输出变量和数个预测变量之间2。

> 2这称作多元线性回归。

到底是算法还是模型？

在本章的开始，我们就两者的差别做过说明。从定义上来说，两者是完全不同的，然而在日常使用时却常常可以交替使用，这给大家带来了些许困扰。严格来说，算法是完成某项任务时需要遵循的一组规则或步骤，而模型是对世界一种附有假设的描述。这两个概念看起来显然是不同的，它们的区别也应该是显而易见的。然而，现实来看并非如此。比如，回归可以是一个统计学模型，也可以是一种机器学习算法。我们觉得，要精确区分两者之间差别，是在浪费时间，毫无必要。

从某种程度上说，这是个历史遗留问题。统计学和计算机科学一直在并行发展，他们常常使用不同的词汇描述同样的东西。这也就导致很难确定某个概念到底是机器学习算法还是统计模型。有一些方法（比如下一节要讨论的k均值）我们称为算法，从统计学的角度来看，它又是一种特殊的高斯混合模型。

因此我们建议，当人们谈起这些方法时，既可以说是算法也可以说是模型，尽量不要让这些干扰到你。（其实在这一行混这么久了，我们也时常对此感到困惑。）

输出变量和预测变量之间存在线性关系是一个大胆的假设，同时也是一个最简单的假设。从数学表示形式来看，线性函数比非线性函数更加基本。在解决问题的时候，我们总是从最简单的模型开始着手，线

性模型是个不错的选择。

虽然简单，但是线性假设有时候也不无道理。有时候，一个变量的变化和另一个变量的变化的确是线性的。比如，你卖的伞越多，你赚的钱越多。此时，你可以充分相信自己做出的线性假设是合理的。而有时候，确认变量之间的线性关系是困难的。但是微积分的角度来说，只要函数是连续的，函数可以被一些局部线性的函数所拟合。因此，局部线性假设大多数情况下都是合理的，但是全局线性假设就很难说了。

线性模型可能适用于类似下面的一些问题：比如说你正在研究一个公司的销售额和该公司在广告上的投入之间的关系，或者某人在社交网站上的好友数量和他每天在该社交网站上花费的时间之间的关系。这些问题的输出变量都是数值型的，也就意味着线性回归模型可能是一个不错的选择。最起码可以说，我们从线性模型作为研究的起步是没有太大问题的。

理解线性回归的一个切入点是先来确定那条直线。我们知道，通过斜率和截距就可以完全确定一条直线$y = f(x) = \beta_0 + \beta_1 x$，但是，这里的设定是完全确定的。3

> 3这样的确定型函数（模型）对于数据分析来说起到了方向性的作用，但是数据往往是有很大噪声的，这就需要在确定型函数的基础之上考虑噪声，也就是随机性的存在，见下文。

考虑随机函数，即便是对于数学功底不错的我们来说，也是一个新鲜的概念，但是在数据科学中却再常见不过了。然而，随机函数从根本上来说，还是建立在确定型函数的基础之上的。因此，我们不妨先讨论一下确定型函数的概念，用几个例子来切身感受一下。

例子1：确定型函数关系 假设你是某个社交网站的创立者，这个网站对所有的会员都收取每月25美元的会费，而这笔钱是你唯一的经济来源。每个月你都会搜集关于会员数和网站利润的数据并持续了两年的时间。你把这些数据都录在了一个表格当中。从数据的表现形式来说，这些数据可以表示成一个个数据对的形式（用户数，利润值），比如下面就是从数据中摘取的前四个数据点：

$$s = (x, y) = (1, 25), (10, 250), (100, 2500), (200, 5000)$$

如果你只把这4个数据点给你的任何一个朋友看，即便这个朋友根本不知道你在做社交网站以及你是如何收费的（这样的朋友绝交了也

罢），他们也能一眼看出数据点钟蕴含的数学模型，如果用y表示理论，x表示用户数，则$y = 25x$。因为其中的模式太过明显，只需要在脑子里面一过就可以迅速得到下面三个结论：

- 两者的关系是线性的；
- 线性关系的强度值是25；
- 应该是一个确定型的关系，最起码从前4个点来看没有例外。

为了再次确认自己的判断，你可以用散点图画出这4个点，其中的关系就一目了然了：的确，是一条严格的，确定型的线性关系。参见图3-1。

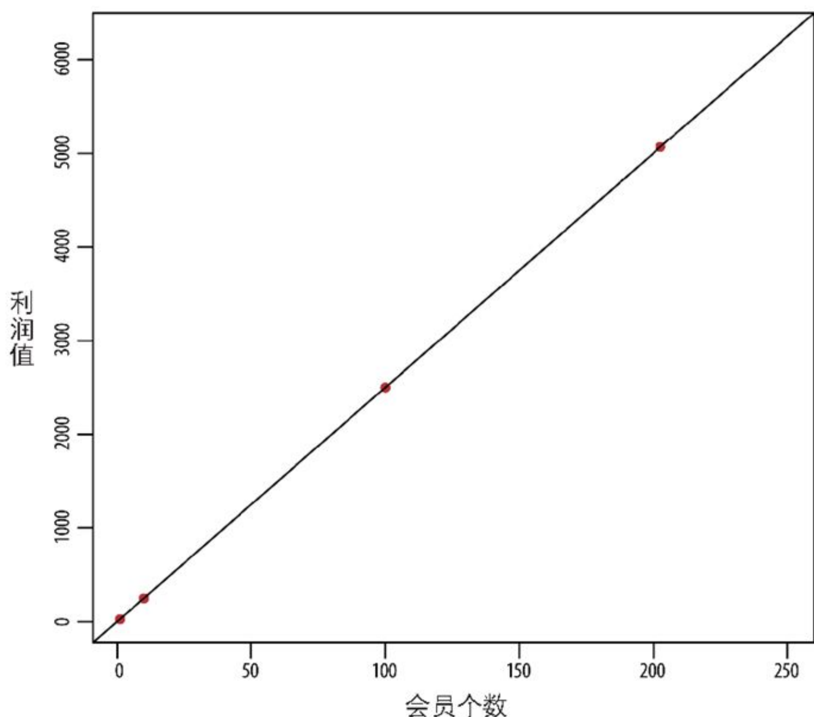


图3-1：一个严格确定型的线性关系图

例子**2**：用户使用数据 假设你有一个用户数据：数据的每一行代表一个用户，数据的每一列是用户在社交网站上一段时间的某个行为变量。假设数据已经被清理过了，总样本量多达几十万。数据的变量包括“总好友数”“本周新好友数”“总访问量”“总浏览时间”“下载的程序数”“被展示的广告数”“性别”“年龄”等。在探索性分析阶段，你可能只

需要从所有的用户中随机抽取100个样本即可。为了探索变量之间的相互关系，可以用类似图3-1那样散点图的方法。比如这里我们关心的目标变量Y = 总浏览时间（单位为秒），预测变量X = 新的好友数。从商业模式的角度来看，如果你的商业模式是卖广告，那么很明显新用户数越多越好，你会承诺给登广告的客户最低的新用户数，这时候就要用到预测模型了，因为你希望能够提前数天或者数周知晓可能的新用户数量。这里我们先把问题简化，看一下两个变量之间的关系。看一眼随机抽取的100个数据，前6行是这样的：

```
7， 276
3 ，43
4， 82
6， 136
10， 417
9， 269
```

乍一看，不像之前的例子，我们已经很难一眼看出两个变量之间的函数关系了（即使你有读统计学的朋友，他们也未必能一眼看出）。因为实际上还有很多数据，所以这个时候靠拍脑袋是没有用的，不妨把它们的关系散点图画出来，如图3-2所示。

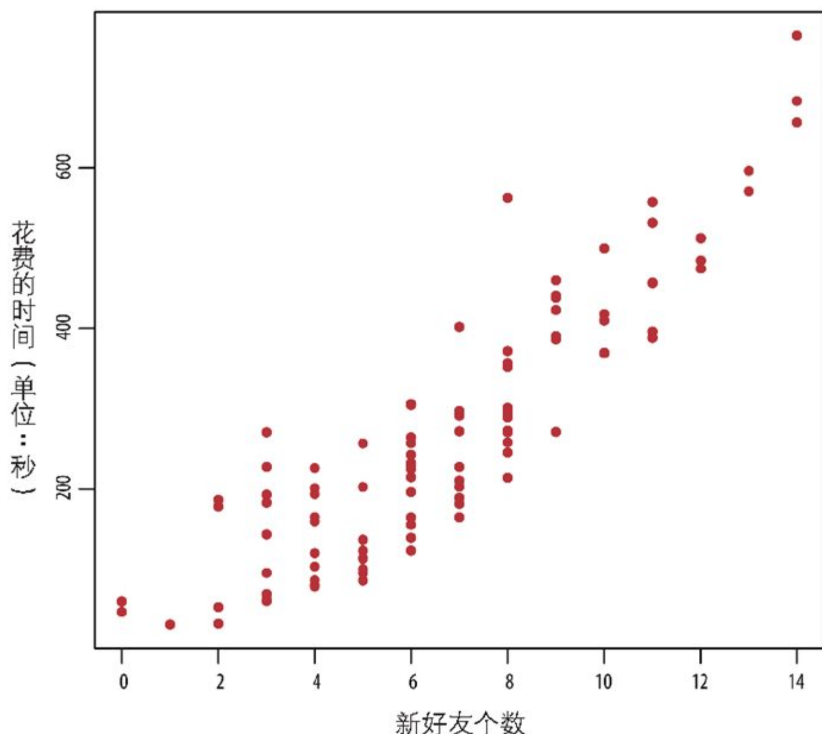


图3-2：看似线性关系的散点图

从散点图来看，这两个变量之间有着某种线性联系。这个结论是合情合理的，因为如果你的新好友数越多，你很可能会花更多的时间在网站上。但是这里的线性联系不能用一个确定型的函数来表示，因为很明显不是所有的点都在一条直线上。但是即便如此，我们还是可以说，两个变量之间的线性关系是存在的：*X*变量的增加同时也带动着*Y*变量的增加，增加的趋势倾向于是一条直线；反之亦然。

小贴士

模型对于数据来说，主要是用来捕捉其中两个方面的信息：第一个是趋势（trend），第二个是变动幅度（variation）。我们先从趋势说起。

首先我们假设，*X*变量和*Y*变量之间的关系确实是线性的，于是我们会尝试在其中画一条直线。

有很多条直线看起来都有可能是不错的选择，我们在其中画了几条，

如图3-3所示。

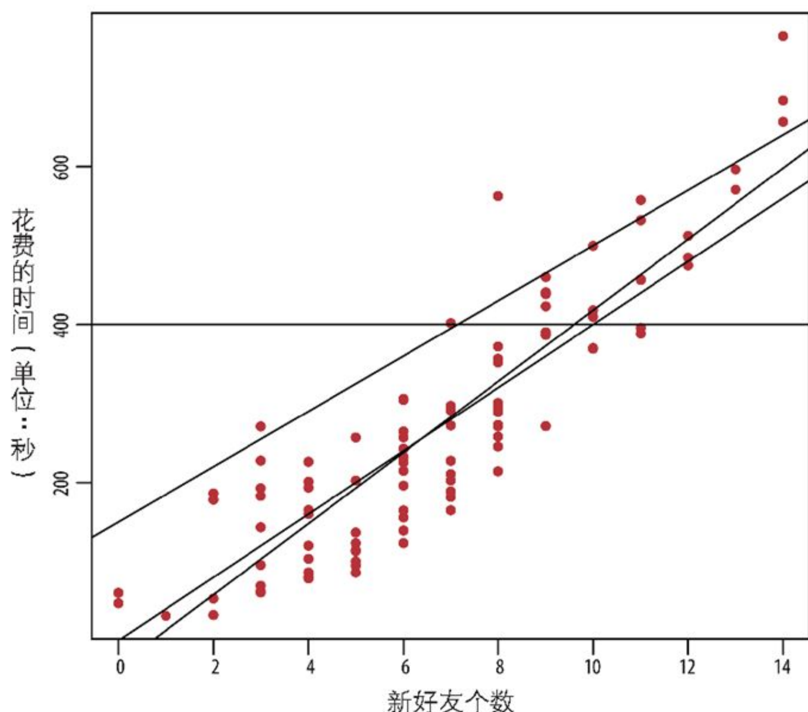


图3-3：哪条直线最佳呢

这么多直线，我们到底应该选择哪一条呢？

因为我们假设了两个变量之间的关系是线性的，因此模型的形式可以表示为：

$$y = \beta_0 + \beta_1 x$$

那么接下来的工作就是，在给定数据样本 $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$ 的情况下，确定最佳的截距（β_0）估计值和斜率（β_1）估计值。

线性模型也可以用矩阵表示：

$$y = x \cdot \beta$$

其中*x*是数据矩阵，β是参数向量。我们的任务就是，找一条最佳的直

线（参数向量估计值）拟合数据。

模型拟合

那么参数向量β到底如何估计呢？一个直观的想法是，如果存在一条最佳拟合的直线，那么所有样本数据点到这条的直线的距离应该是所有直线中最小的。

很可能会有很多线看起来都拟合得不错，但是最优的只有一条。可能有很多种定义“最优”的方式，对于线性回归来说，这里的“最优”指的是距离最小化。那么“距离”在这里又是什么意思呢？

我们用图3-4为大家讲解一下“距离”的概念。假设数据点的y值用y_i表示，其在直线上的拟合值（预测值）为$\hat{y_i}$, 那么一个样本点与其拟合值的距离可以定义为两个点在y值上的“离差平方”：$(y_i - \hat{y_i})^2$。所有数据点的距离之和也称作“离差平方和”：$\sum_i (y_i - \hat{y_i})^2$。最优的那条直线具有最小的“离差平方和”。可以看出，这里距离的定义，也就是“离差平方和”还可以解释为模型的预测误差。这样的估计方法就是著名的最小二乘估计法。

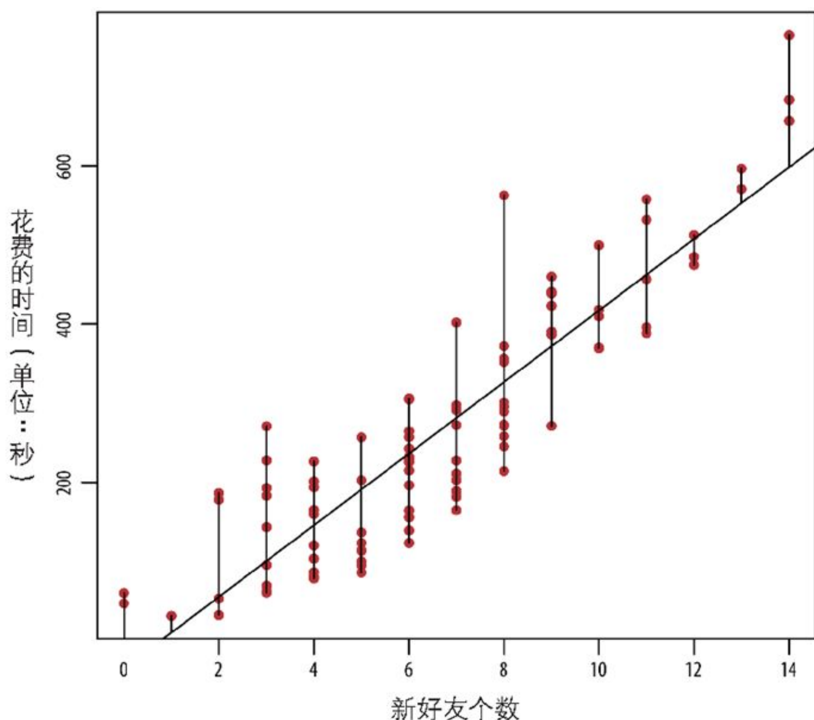


图3-4：具有最小“离差平方和”的那条最优拟合直线

离差平方和表示为RSS（Residual Sum of Squares），可以表示为：

$$RSS(\beta) = \sum_i (y_i - \beta x_i)^2$$

*i*代表某个数据点。离差平方和是一个有关于β的函数，而为了找到最优的β，我们需要最小化离差平方和。

微积分告诉我们，对RSS(β)针对β求导并令其为0即可找到可能的最优解。$RSS(\beta) = (y - \beta x)^t(y - \beta x)$，可以得到：

$$\hat{\beta} = (x^t x)^{-1} x^t y$$

$\hat{\beta}$代表β的估计值，真实的β是无从得知的。在得到β估计值的表达式之后，只要将观测数据的值代入即可计算出实际的估计值。

在R软件中拟合一个线性模型再简单不过了，假设有一列数据代表因变量*y*，一列数据代表自变量*x*，则拟合的R代码为：

```
model <- lm(y~x)
```

假设真实的数据真像我们之前展示的那样，其前6行为：

```
x,  y
7, 276
3, 43
4, 82
6, 136
10, 417
9, 269
```

那么用下面的R代码：

```
model <- lm(y~x)
model

Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)           x
-32.08              45.92
```

```
coefs <- coef(model)
plot(x, y, pch=20,col="red", xlab="Number new friends",
     ylab="Time spent (seconds)")
abline(coefs[1],coefs[2])
```

因此，模型最终的最优估计直线为：$\hat{y} = -32.08 + 45.92x$，当然也可以简化为$\hat{y} = -32 + 46x$，拟合的直线效果见图3-5的左半部分。

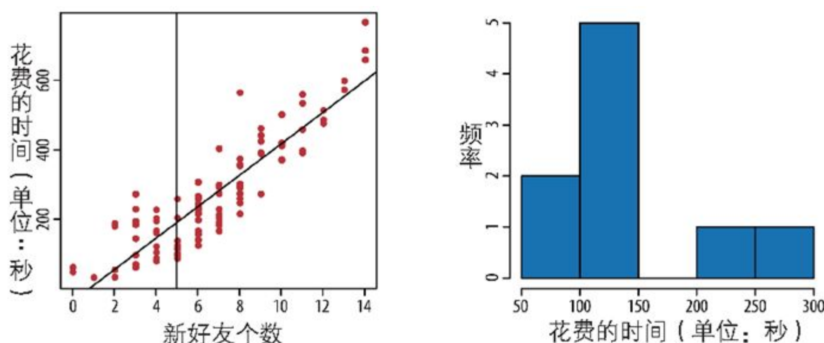


图3-5：左图是线性回归的最优拟合直线。可以看到，对于任意固定的x值，比如$x = 5$，相应的y值在观测数据层面是可变的。对于$x = 5$，我们在右图画出了相应y预测值的可能分布

至于到底是否采纳这个线性模型，将它用于数据关系描述和结果预测，这取决于数据科学家自己的判断。如果新数据的x值为5，代表某人的新好友数为5，那么根据模型，y的预测值为-32.08 + 45.82 * 5 = 195.7（秒）。一个自然的问题是，对于得到的预测值，我们有多大的自信认为它会十分接近真实值呢？

这在统计学上叫作置信值的问题，解答它需要将模型的内涵稍作延伸。可以想象，如果用户的新好友数为5，那么这些用户在网站上花费时间的预测值不可能只是一个定值195.7秒，一个合理的情况是这些用户花费的时间都在195.7秒附近波动。因此，线性模型得到的预测值只是所有可能预测值的一个总体趋势，而围绕这个趋势的波动性还没有被模型考虑进来。

最小二乘模型的延伸

我们刚刚讨论的是一个简单线性回归模型（一个输出变量，一个预测变量），模型参数的预测采用了最小二乘法来估计β。在此模型的基础上，我们可以主要从三个方面加以延伸：

1. 增添一些关于模型误差项的假设；

2. 增添更多预测变量；

3. 对预测变量加以变换。

增添关于模型误差项的假设

如果你只是应用线性模型预测给定*x*值情况下的*y*值，那么得到的预测值只是一个确定值，这就忽视了预测必然存在的可变性。图3-5（右）就很好地说明了这个问题，对于一个给定的$x = 5$的预测情形，*y*的值是不定的。为了让模型捕捉数据中的不确定性，可以将模型的形式扩展为：

$$y = \beta_0 + \beta_1 x + \epsilon$$

其中的ϵ是模型中的新加项，也称作“噪声”项，代表数据中不能被模型部分拟合的部分。它也称作误差项——ϵ代表模型的实际误差，也就是实际观测值与真实回归直线上所得值的差距。真实的回归直线永远是未知的，而你只能通过$\hat{\beta}$估计。

我们通常假设该残差项服从一个均值为0，方差未知的正态分布，故：

$$\epsilon \sim N(0, \sigma^2)$$



对残差项的正态分布假设有时候是不合理的，比如说当数据具有明显的“厚尾分布”（fat-tailed distribution）特征时，或者模型的主体部分只能拟合数据中的一小部分特征时。在金融数据建模中，这都是经常发生的，因此正态分布的假设很难适用于对金融数据的建模。

但是，这也并不代表我们在金融数据研究中完全摒弃线性回归模型（及其误差项的正态分布假设），只是说金融数据中的厚尾特征对传统的线性回归模型提出了挑战。

在误差项的正态分布假设下，我们可以从条件分布的角度解释线性回

归模型。也就是说，对于给定的x，y的条件分布是一个正态分布：

$$p(y|x) \sim N(\beta_0 + \beta_1 x, \sigma^2)$$

回到刚才$x = 5$的情形，也就是说，对于所有新好友数为5的用户，他们在社交网站上花费的时间服从一个正态分布，该分布的均值为$\beta_0 + \beta_1 \times 5$，方差为$\sigma^2$。$\beta_0, \beta_1$以及$\sigma^2$的值需要从数据中估计。

那么现在的问题是，到底如何拟合这个模型呢？到底如何估计这些参数值呢？



其实，可以从数学上证明，无论误差项的分布形态如何，最小二乘估计都具备同样优良的性质：它是无偏估计量，并且具有最小的估计方差。若想了解该性质以及其中数学证明的细节，我们建议大家找一些有关统计推断的书读一读，比如Casella和Berger合著的《统计推断》（*Statistical Inference*）。

β_0和β_1的估计方法我们已经探讨过了，是基于最小二乘估计的；因此难题是到底如何估计σ^2。基本的想法是用实际的观测误差，也称作残差，去估计实际的误差。实际残差为:

$$e_i = y_i - \hat{y}_i = y_i - (\hat{\beta}_0 + \hat{\beta}_1 x_i), i = 1, ..., n$$

σ^2的无偏估计量为：

$$\frac{\sum_i e_i^2}{n-2}$$



看到上面的公式，好奇心重的同学肯定会问：为什么上式中的分母是n - 2呢？这是因为，如果用n - 2作为分母，而不用n，那么这个误差估计量在统计学上来说称作一个无偏估计量。n - 2中的2是

模型中参数的个数。上面提到过的Casella和Berger合著的《统计推断》一书有详细的背景知识介绍，感兴趣的同学不妨翻看一下。

上面的方差估计量也叫作均方误差（mean squared error），它衡量的是预测值偏离实际观测值的程度。对于预测问题来说，均方误差是被广泛使用的一个误差估计量。对于回归问题来说，它可以作为一个方差估计量，但是对于其他的问题，我们可能不能简单地把它解释为此。接下来我们还会反复提到均方误差的概念。

模型评估标准

我们之前问过类似的问题：对于模型参数的估计值，我们有多大的信心它们是准确的呢？从R的模型输出来看，我们可以依赖p值和R方这两个统计量。在R中，在拟合了一个模型之后，如果在控制台中输入**summary(model)**，那么会得到以下输出结果：

```
summary (model)
    Call:
    lm(formula = y ~ x)
    Residuals:
    Min      1Q  Median
                             3Q      Max
    -121.17  -52.63   -9.72   41.54  356.27
    Coefficients:
                Estimate Std. Error t value Pr(>|t|)
    (Intercept)  -32.083     16.623   -1.93   0.0565 .
    x             45.918      2.141   21.45   <2e-16 ***
    Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05             '.' 0.1
    Residual standard error: 77.47 on 98 degrees of
    Multiple R-squared: 0.8244,    Adjusted R-squared:
    F-statistic:   460 on 1 and 98 DF,  p-value: < 2.2e-16
```

***R*方**

$R^2 = \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2}$，这是*R*方的定义公式。可以解释为数据中能够被模型所解释的方差占数据总方差的比重。将这个公式与之前均方误差的公式对比，我们会发现，均方误差实际上是*R*方公式中分式的分子部分，分母对应的是数据中的总方差，因此整个分式代表的是数据中未被模型解释的方差的比重。

***p*值**

在*R*的模型输出结果中，模型参数的估计是在Estimate那一列中显

示。要想得到每一个参数估计的p值，我们要计算$Pr(gt; |t|)$。对p值的解释我们之前略有提及：我们先设定了一个原假设（null hypothesis），假设参数值β为0。对于每一个β，p值代表的是在原假设的基础之上我们可以得到观测数据的概率，也就是在原假设的基础上得到相应t统计量值的概率。这也就意味着如果p值很小，那么在相应的原假设下得到观测数据的概率就很小，因此原假设很可能是不正确的。也就是说，β应该显著得不为0。

交叉验证

还有另外一种评估模型的方法，它大概要遵循这样的程序：分割数据，80%用作训练数据集，20%用作测试。在训练数据及上拟合模型并估计模型的参数，并在测试数据集上计算出模型的均方误差，并与训练数据集上模型的均方误差进行比较。如果两种均方误差的差异很小，那么可以认为模型具有不错的扩展性，其过拟合的风险较小。我们也建议大家尝试改变一下训练数据集的大小，看看两者之间有何联合变动关系，这样的模型验证过程叫作“交叉验证法”，如图3-6所示。

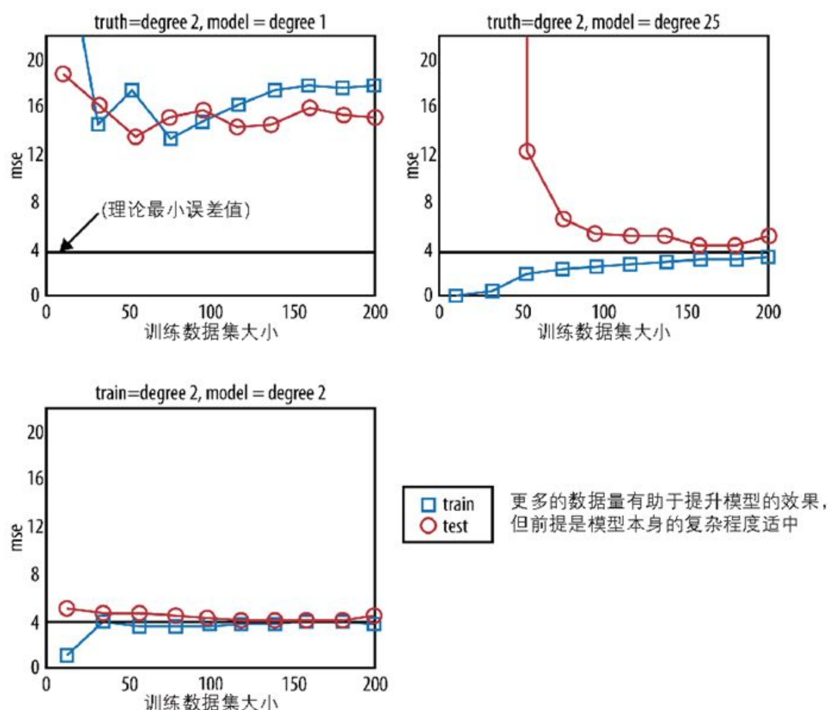


图3-6 在训练数据集和测试数据集上比较均方误差。该图截自Nando de Freitas教授的课件。这里，由于数据是从一个已知的分布模型中模拟得来的，因此真实的

模型误差是事先知道的

其他类型的模型误差测度

均方误差属于“损失函数”的一种，在线性回归中均方误差是最常使用的误差指标，它能够很好地测度模型的拟合效果。它的另外一项优点就是，如果假设误差项是正态分布的，那么模型的估计可以完全依赖最大似然函数法。其他类型的模型误差测度还有很多，比如基于绝对值误差。人们甚至可以根据研究的目的和个人的偏好设计和使用相应的误差测度指标。但从目前来看，我们还是觉得均方误差是个不错的选择。

增加预测变量 我们刚才讨论的是最简单的线性回归模型：一个自变量以及一个因变量。在此模型的基础之上，我们可以增加预测变量（自变量）的个数，从而得到多元回归模型：

$$y + \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \epsilon.$$

之前的模型表达式在这里也同样适用，因为我们之前用了矩阵表达式，增加一个变量无非是在矩阵上增加一列而已，模型的表达形式不会改变。还以刚才的例子来说，对于预测人们在网站上所花去的时间，人们的年龄和性别也可以作为预测变量。至于在线性回归模型中到底应该选用哪些变量，我们将在7.4节详细介绍。对于上面的具有三个预测变量的模型，在R中的模型拟合代码为：

```
model <- lm(y ~ x_1 + x_2 + x_3)
```

若要在模型中添加一个交叉变动项也很简单：

```
model <- lm(y ~ x_1 + x_2 + x_3 + x_2*x_3)
```

在确定到底使用哪些预测变量时，散点图通常是一个很好的工具。我们可以画出预测变量与每一个自变量之间的散点图，以及自变量之间的散点图，以找到或确认可能有用的预测变量和变量之间的交叉变动关系。基于预测变量的每一个值，画出因变量的条件分布（$y|x$）直方图也会有所帮助。在选定了预测变量之后，同简单线性回归模型一样，我们仍然可以使用R^2、p值和交叉验证的方法评价模型的质量。

变换 刚才的模型假设了y与x之间的线性关系，然而为什么我们不能使用x的高阶项呢，这样我们就可以得到一个类似下式的多项式回归模型：

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3$$

我们知道，多项式回归模型不是线性回归模型。但是如果把x的高阶项看成新的变量，上式的表达形式仍然保持着“线性”的模样。也就是说，我们可以把多项式回归模型看成线性模型，前提是把变量做一下适当的变换。比如，我们可以假设$z = x^2$，$w = x^3$，这样上面的模型就是一个基于三个预测变量（x, z, w)的线性回归模型了。这样的变量变换在线性回归中经常会用到，其他的变换方法还包括取对数、离散化、二元化等。

但是，是否对一个变量进行变换是建立在事实基础之上的，比如说在预测人们在网上花费的时间这一问题上，其中一个预测变量就是好友的个数。通过散点图，如果我们发现好友的个数与花费时间的关系呈现明显的曲线特征，而非直线，那么我们就应该考虑对“好友的个数”这个变量做适当的变换，以满足线性回归模型的“线性”要求。

我们现在所讨论的各种有关模型形式和变量变换的问题，是建模工作者必须要直面的一个最重要的挑战之一：那就是你永远不知道真理是什么。很可能真实的模型是二次型的，而你最后还是使用了线性模型。有许多的工具可以帮助我们尽量客观地确定模型的形式，但你永远不能100%地肯定你的模型是正确的。数据量越多往往对于建模越有帮助，我们可以无限地接近真理，但永远无法到达那里。

回顾一下

现在让我们回顾一下我们在构建模型和拟合模型的时候做过的诸多假设：

- 线性假设；
- 误差项是正态分布的，并且均值为0；
- 误差项是相互独立的；
- 误差项具有恒定的条件方差；
- 预测变量都是有用的。

我们应用线性回归模型，主要基于以下两个目的：

- 用作预测，在知道自变量值的情况下，预测因变量的可能值；
- 用作变量关系的解释，比如确定两个变量之间可能的相关关系或者联合变动关系等。

练习

我们可以在R中用模拟的方法帮助理解和探索这些新概念。模拟的好处在于，你永远知道数据是如何产生的，因此也就知道真实的模型是什么样子。换句话说，你就是“上帝”，真理是掌握在你自己手中的。因此，你可以评判一个模型的效果到底好还是不好。

在模拟的数据上确认完模型的有效性之后，就可以把目的转移到真实的数据上了。接下来我们会展示如何模拟出一个数据集，以及如何利用模拟数据集帮助我们探索和理解模型：

```
# 模拟一个虚拟数据集
x_1 <- rnorm(1000,5,7) # 从一个均值为5，标准差为7
                       # 的正态分布中随机模拟1000个值
hist(x_1, col="grey")  # 画出p(x)
true_error <- rnorm(1000,0,2)
true_beta_0 <- 1.1
true_beta_1 <- -8.2
y <- true_beta_0 + true_beta_1*x_1 + true_error
hist(y) # 画出 p(y)
plot(x_1,y, pch=20,col="red") # 画出p(x,y)
```

1. 在模拟数据上拟合线性回归模型，比较参数β的估计值与真实值的差距。

2. 模拟一个新的服从伽马分布的变量x_2，并构建一个新的y变量：$y = x_1 + x_2$。接下来可以做很多事，我们可以拟合一个只有x_1,或者只有x_2的模型，再拟合一个包含两个变量的模型。随后，我们可以在不同样本量大小的数据集上拟合模型，并可以通过交叉验证的方法看模型的均方误差在训练数据集和测试数据集上的表现。

3. 在模型中加入新的预测变量z，$z = x_1{}^2$。比较模型在加入该变量前后的拟合效果有何不同。通过交叉验证和改变样本量的方式，比较模型在训练数据集和测试数据集上的不同表现。

4. 还有很多东西可以玩：(a)改变真实的参数值；(b)改变模型误差项的分布类型；(c) 在模型中加入更多具有不同分布类型的预测变量。（在R中，`rnorm()`函数的作用是从某个正态分布中生成一些随机值。顾名思义，`rbinom()`函数是从二项分布中产生随机值。可供尝试的分布类型非常多，大家可以多尝试几种。）

5. 画出所有变量之间关系的散点图，以及单个变量的直方图。

3.2.2 *k*近邻模型（k-NN）

k近邻模型是一个分类算法，在给定一个已经做好分类的数据集之后，k近邻可以学习其中的分类信息，并自动给尚未分类的类似数据分好类。

分类是建模的基本问题之一，我们可能想把数据科学家分成两类：“性感类”和“不性感类”， 或者把人分成两类：“高信用度”和“低信用度”；酒店则可能按星级分成5类：“五星级”“四星级”“三星级”“二星级”和“一星级”。对于酒店，也许我们可以多加一类，将那些非常差，连“一星级”都不够格的酒店分类为“零星级”。对于患者，我们可能希望将他们分为“高患癌风险病人”和“低患癌风险病人”两类。分类的任务在我们日常生活中可以说无处不见。

现在让我们停下来思考一下，对于分类问题我们可以应用线性回归模型吗？

答案是：不一定，也不是没有可能。从模型形式上来说，线性模型的输出值是连续性的实数值，而分类模型的任务要求是得到分类型的模型输出结果。从这一点上来说，线性模型是不适用于分类问题的。

但是，如果我们换一个思路，问题还是可以解决的。这里要用到“阈值”的概念。也就是将连续性数值离散化。比如，如果我们想根据人们的年纪和实际收入预测他的信用值，那么我们可以选取700作为阈值，如果其信用预测值高于700，则将其列为“高信用度”类。如果其信用预测值低于700，则将其归类为“低信用度”类。这里我们用一个阈值将一个连续性变量转换成了一个二元变量，我们当然可以使用更多的阈值将预测性变量离散化成具有更多类别的变量。比如，可以用4个阈值将信用度分成5个类别：极低信用度、低信用度、中级信用度、高信用度和极高信用度。

然而，并不是所有的变量都像信用度一样，可以被阈值分为几个严格不重叠的类别。有些类别可能是模棱两可的，比如一个人的政治倾向，他可能是“民主党”，也可能是“共和党“，或者可能是政治中立的。这样的具有概率特性的分类变量，我们该如何分析呢？

k近邻的主要想法是，根据属性值的相似度找到某个对象的相似对象们，并让其相似对象们一起“投票”决定该对象到底应该属于哪一类。如果有某两个或者更多个的类别投票数相同，那么就从这些类别中随机挑选一个作为该对象的类别值。

让我们举一个例子。假设一个人会对自己看过的每部电影都打上“好看”或者“不好看”的标签，现在根据这些历史标签信息，如果给定一部

新的电影叫作《狂野的数据》，我们可以用k近邻的方法预测此人是否会觉得这部电影好看。方法首先是要确定电影的属性值，包括电影的长度、风格、性爱场景数、奥斯卡最佳演员的个数以及电影的拍摄预算等。根据这些属性，我们可以找到与《狂野的数据》最相似的k部电影，并记录下这k部电影的标签信息。如果这些电影总体来看包括更多的是“好看”的标签（这个过程，叫作“投票”），那么我们也自然会预测《狂野的数据》这部电影会得到此人的青睐。

k近邻方法需要解决两个核心问题：其一是如何根据属性定义个体之间的相似性或者紧密程度。一旦有了明确的定义，我们就可以把某个带预测个体的“近邻”们找出来，让它们投票决定该个体的类别属性。

这就自然引申出了第二个问题：到底如何才能确定一个最优的k值呢？对于数据科学家来说，k值的确定十分关键。接下来我们会详细讨论解决方法。

首先让我们看一个现实生活中的例子。

信用评分实例

设想手边有一个关于人们信用评分的数据集，包含的变量有人们的年龄、收入以及信用评分的等级。信用等级变量只包含两个类别，分别是“高信用度”（high）和“低信用度”（low）。给定这个数据，我们想据此预测一个新人的信用等级。

下面是这个数据集的前几行，收入在第二列，其单位是千美元：

```
age income credit
69  3       low
66  57      low
49  79      low
49  17      low
58  26      high
44  71      high
```

图3-7是一个特别的散点图，横轴是年龄变量，纵轴是收入变量，实心点代表高信用度，虚心点代表低信用度。

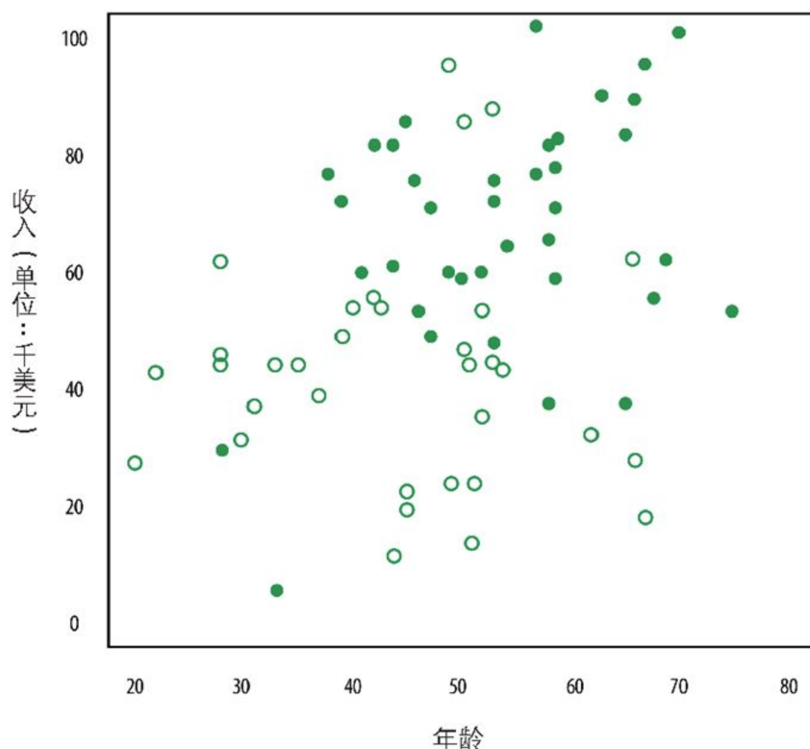


图3-7：信用等级关于年龄和收入变量的散点图

现在有一个预测问题：根据刚才的数据集的信息，如果告诉你现在有位新人，他的年龄是57岁，收入是37000美元，那么他的信用等级应该是什么呢？或者说，他的信用等级更可能是高还是低呢？如图3-8所示，问号的位置代表新人的两个属性（年龄和收入）的位置，根据他周边的标签信息，*k*近邻方法可以告诉我们他的信用等级更应该是高还是低。

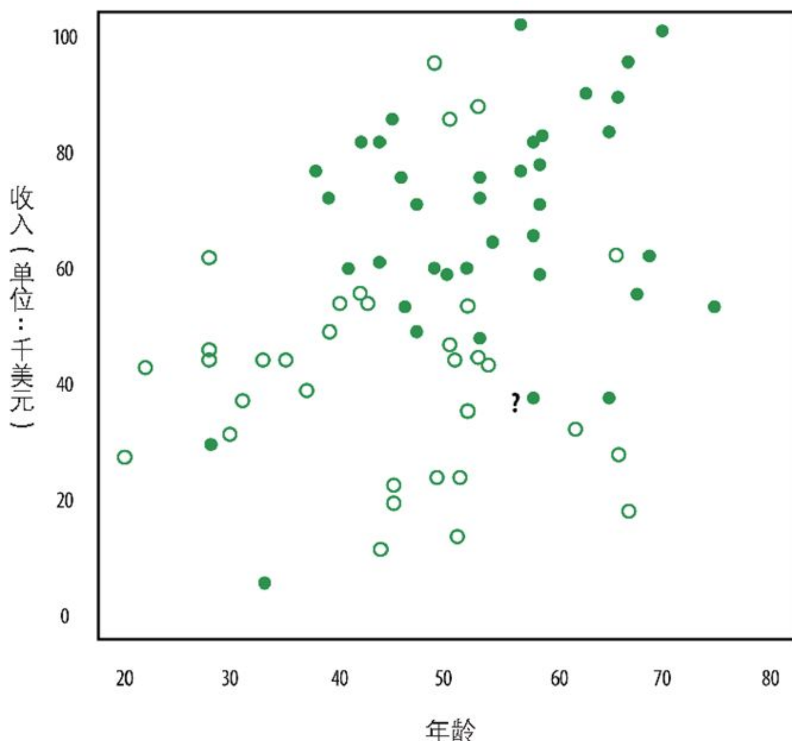


图3-8：这个新人的信用等级是高是低呢

下面我们就列出k近邻方法工作的详细步骤。

1. 首先确定相似性定义，这通常也叫作“距离”。

2. 将数据分割成训练数据和测试数据集。

3. 选择一个模型评价标准。（对于分类问题来说，误分率是个不错的选择，我们会在稍后章节详细讨论。）

4. 选择不同的*k*值，并应用*k*近邻模型，看哪个模型的效果最好。

5. 基于选定的模型评价标准，选出最优的*k*值。

6. 选定*k*之后，就可以做样本外测试了。我们刚才提到的一个新人就是一个样本外的例子。

相似性/距离测度

相似性测度的选择在很大程度上要取决于问题的背景和数据本身的特征。举个例子来说，对于社交网络数据来说，如何衡量两人之间的“相似性”呢？一个通常的做法是用两人之间共同好友的个数来衡量。而这样的测度方式放在别的数据上可能就不太合适了。

对于我们刚才讨论的问题来说，如果变量取值大致都在一个水平上，那么二维平面上的欧几里得距离是个不错的选择。当然，我们很难假设所有变量的取值都在一个水平上，很多时候，变量的取值范围会差别很大。

> 注意：前方有坑！
>
> 对于建模来说，变量的取值水平是个很重要的问题，如果处理得不好，很可能会影响整个模型的效果。
>
> 让我们来看一个例子：年龄通常的取值单位是年，收入是美元，而信用评分的取值往往是由权威部门规定的——就像SAT分数一样。因此一个人的观测数据可以用一个三元向量表示，如（25, 54 000, 700）和（35, 76 000, 730）。因为收入的取值水平最高，因此在计算距离的时候，相似性的测度很可能会被收入这一个变量所主导，而其他两个变量很难在该测度中体现出来。这就是变量取值水平的问题，我们总是希望变量的取值都大致在同一个水平上。
>
> 然而，如果收入是用“千美元”来度量，那么相应的三元变量会变为（25, 54, 700）和（35, 76, 730）。如此一变，三个变量的取值就大概在同一水平上了。
>
> 总而言之，变量的计量方式以及变量之间距离的定义方式，在统计学中我们叫作“先验信息”，它们都可能会影响到模型最终的效果。

欧几里得距离又叫欧氏距离，对于在实数轴上取值以及能够在平面或者多维空间中描绘出来的变量来说，它是一个不错的距离量度。

- 余弦度（**Cosine Similarity**）

  余弦度同样可以用于度量两个实数向量$\vec{x}$和$\vec{y}$之间的相似程度，而且余弦度是一个介于-1和1之间的数。如果取值为-1则代表完全不同，1代表完全一样，而0则代表两个向量之间相互独

立。回顾一下余弦度的定义：$\cos(\overrightarrow{x}, \overrightarrow{y}) = \frac{\overrightarrow{x} \cdot \overrightarrow{y}}{\|\overrightarrow{x}\| \, \|\overrightarrow{y}\|}$。

- **Jaccard**距离或相似度

  Jaccard距离通常用作衡量两个集合之间的相似度——比如说Cathy的朋友集合表示为**A** = {Kahn, Mark, Laura, ...}，Rachel的朋友集合表示为**B** = {Mladen, Kahn, Mark, ...}，那么这两个朋友集合的Jaccard相似度为$J(A, B) = \frac{|A \cap B|}{|A \cup B|}$。

- **Mahalanobis**距离

  也称作马氏距离，同样适用于两个实数向量。相比于欧氏距离，马氏距离考虑了两个变量之间的相关关系，并且不用担心变量取值水平的问题。马氏距离的定义为：$d(\overrightarrow{x}, \overrightarrow{y}) = \sqrt{(\overrightarrow{x} - \overrightarrow{y})^T S^{-1} (\overrightarrow{x} - \overrightarrow{y})}$，其中S是两个向量间的协方差矩阵。

- **Hamming** 距离

  Hamming距离主要用来衡量两个字符串，字组或者具有相同长度的DNA序列之间的相似程度。比如，单词olive和ocean的Hammning距离为4，因为除了第一个字母o相同之外，这两个单词包含的其他四个字母都不相同。同理，单词shoe和hose之间的Hamming距离是3，这是因为除了最后一个字母e相同之外，其他三个字母在对应位置上均不相同。因此，Hamming距离的计算方式其实十分简单，按照位置顺序比对两个单词字母之间是否相同，每个位置上字母的不同，相应Hamming距离的值都增加1。

- **Manhattan**距离

  Manhattan距离（曼哈顿距离）也用来测度两个*k*维实数向量之间的距离。之所以叫作Manhattan距离，是因为要把这个距离想象成城市中两点之间，如果用计程车行走的路线来算的话，应该为多长。（在城市中行走或者行车需要遵循一定的路线规定，譬如大型建筑要绕行，不能穿墙而过。）曼哈顿距离的定义为：$d(\overrightarrow{x}, \overrightarrow{y}) = \sum_i^k |x_i - y_i|$，*i*代表向量中元素的位置。

除了上述的这些距离测度指标，当然还有很多别的选择。至于到底选择什么样的指标，这要取决于研究的问题和数据本身。当你觉得困惑

的时候，不妨先到谷歌上搜索一下，看看别人是怎么用的。

有一种情形会使得问题变得更加复杂，那就是当数据有很多不同类型变量的时候。比如说，在电影评分数据库中，有些变量是数值型的（比如电影预算和演员数量等），而有些是分类型的变量（比如电影的风格）。在选择距离测度的时候，我们还要根据变量类型的不同而有所变通。但无论如何，距离的测度选择都具有相当的自由和主观性。

我们甚至可以自己定义距离的测度方式，比如对于电影评分数据来说，我们可以定义，如果两个电影的风格相同，那么它们之间的距离值增加为0，但是如果风格不同，则距离值增加为10。之所以选择10这个数据作为单位增加值，这是因为它和预算规模（百万美元）相称，而预算规模为0～100。至于到底选择10，50还是更大的值作为单位增加值要取决于其他变量的取值水平而定。如果其他变量的取值水平较大，那么很可能50会更好一点。

无论做何选择，我们总是希望这个选择在可能的选择中是最优的，虽然很难做到，但也还是值得一试。从模型验证的角度来说，我们尝试很多的选择，并通过模型验证的方式确认哪一个会是最优的。在刚才的选择中，10相对于k近邻算法本身是第二个需要确定的数值（k是第一个，也是最重要的），我们可以把它作为模型的参数最优化，也可以作为建模之前已经掌握的先验信息。至于到底应该怎么看待它，也同样取决于这个问题本身，研究人员自己看待这个问题的角度以及数据本身的特点等。

训练和测试数据集

对于所有的机器学习算法来说，将数据集分成训练数据集和测试数据集是最为通用的做法。数据集用来“训练”模型，而测试数据用来独立地测试“训练好”的模型在实际问题上的真实表现。

对于k近邻模型来说，训练阶段的数据是包含因变量（信用等级）的标签值的。在模型测试阶段，我们假装不知道因变量的真实标签值，并用在训练阶段得到的k近邻模型预测这些标签值。

要想实现测试的目的，我们必须从原始数据中保留一些数据出来，这些数据不用于模型训练而只用于测试。通常来说，我们会随机抽取20%的数据出来作为测试用数据集。

以下的R代码可供参考：

```
> head(data)
age income credit
69  3        low
66  57       low
49  79       low
49  17       low
58  26       high
44  71       high

n.points <- 1000 # 数据集中的行数
sampling.rate <- 0.8

# 我们需要测试数据集中的行数去计算误分率
num.test.set.labels <- n.points * (1 - sampling.rate)

# 在所有的数据行中，随机地选取一部分用作训练数据集
training <- sample(1:n.points, sampling.rate * n.points, replace=FALSE)
train <- subset(data[training, ], select = c(Age, Income))
# 这些行数就代表训练数据集的规模
# 剩下没有被随机选中的数据行就代表了测试数据集
testing <- setdiff(1:n.points, training)
# 没有被选中的行数就代表了测试数据集的规模
test <- subset(data[testing, ], select = c(Age, Income))
cl <- data$Credit[training]
# 这些是训练数据集的标签值
true.labels <- data$Credit[testing]
# 测试数据集的标签值暂且保留
```

选择一个模型评价标准

到底应该如何评判你所选用的模型（及其参数值）的效果呢？

就像我们之前反复说的，模型的评价是非常灵活和主观的，并没有普适的标准。对于某个实际问题，你可能觉得高误分率的严重性要大于其他的误差测度，或者你会觉得伪阴性的严重性要大于伪阳性。因此，在确定一个模型的评价标准时，你可能要和某个问题领域的专家好好交流一番。

举例来说，如果分类模型的对象是人们是否患癌，那么模型的伪阴性当然越小越好（伪阴性指的是某人实际患癌，但是模型却告诉我们他没有患癌）。在确定合理的模型评价标准之前，我们最好与癌症领域的专家医生做好交流沟通，以更深入地了解手中数据所涉及的实际问题背景。

但是对于伪阴性，有一个极端的问题是：如果你想确保伪阴性为0，你可以把所有的病人都分类为癌症患者。这当然也不合理，因为相应的伪阳性率会增加。在分类问题中，这叫作敏感度（sensitivity）与

特异度（specificity）的权衡。敏感度指的是所有的患癌病人实际被诊断为患癌的概率，而特异度指的是未患癌的病人被诊断为未患癌的概率。理想的模型需要具备高敏感度和高特异度。



关于敏感度和特异度的其他说法

敏感度又叫作“真阳性率”（true positive rate）或者召回率（recall)。来自不同学术研究领域的研究人员会使用不同的术语，但是它们都代表的一个意思。特异度也叫“真阴性率”（true negative rate）。 既然有“真阳性率”和“真阴性率”，当然也有“伪阳性率”和“伪阴性率”，但是后两个术语没有其他的表述形式。

我们也可以用“精确度”（precision）作为模型的评价标准，关于精确度我们会在第5章详细介绍。之所以有些术语在不同的领域有不同的表述形式，是因为关于分类的问题的研究在统计学、机器学习等领域是独自发展的，针对同样的问题也就自然的出现了一些不同表述。比如说，“精确度”和“召回率”这样的术语就来自于信息检索领域，但要注意的是，“精确度”不同于“特异度”。

还有一种模型评价标准叫作“准确度”，指的是所有被正确分类的标签数占总标签数的比例。因此，误分率 ＝ 1 - 准确度。最小化模型的“误分率”就等同于最大化模型的“准确度”。

小结

在介绍完距离的定义和模型的评价标准问题之后，模型的前期准备工作就基本完成了。

对于测试数据集中的每个人，你都先假装不知道他们的真实信用等级。用*k*近邻方法找到与每个人最相似的三个人的信用等级，并用“投票”的方式确认此人的信用等级。在预测完测试数据集中所有人的信用等级之后，与他们真实的信用等级做比较，并计算出测试数据集上的误分率，它就代表该*k*近邻模型的实际预测效果。在R中，只需要一行代码就可以完成这些工作：

```
knn (train, test, cl, k=3)
```

*k*的选择

我们已经说过，*k*对于*k*近邻模型来说是最为重要的参数，至于如何选择一个合适的（或者最优的）的*k*值，这要求你对数据本身的背景有深刻的理解，并且还需要通过不断地尝试（基于选定的模型评价标准）以确定一个合适的*k*值。



二元分类

刚才的信用等级分类是一个二元分类问题，因为待分类的变量“信用等级”只有两个类别“高信用度”和“低信用度”。对于二元分类情况，我们建议选择一个奇数*k*值。原因是，奇数*k*值在“投票”的过程中总会出现一个“多数”类。当然，你也可以选择一个偶数*k*值，如果需要“投票”出现平手的情况，只需要随机挑选一个类即可。

```
# 我们将迭代20个k值，并且看哪个k值对应的误分率最小
for (k in 1:20) {
     print(k)
    predicted.labels <- knn(train, test, cl, k)
   # 这里使用了R中的knn()函数
   num.incorrect.labels <- sum(predicted.labels != true.labels)
   misclassification.rate <- num.incorrect.labels /
                                 num.test.set.labels
    Print(misclassification.rate)
```

下面是上面一段代码4的输出值：

> 4该段代码尝试了20个不同的*k*值，并在测试数据集上评价每一个*k*值的实际预测效果。

```
k  misclassification.rate
1,   0.28
2,   0.315
3,   0.26
4,   0.255
5,   0.23
6,   0.26
7,   0.25
8,   0.25
9,   0.235
```

```
10 , 0.24
```

从测试的输出结果来看，$k = 5$是最优的选择，因为其对应的误分率最低。把$k = 5$的k近邻模型应用到之前提到的新人上，该人年龄是57岁，收入为37000美元。在R中，敲入以下代码：

```
> test <- c(57, 37)
> knn(train, test, cl, k = 5)
[1] low
```

k近邻模型告诉我们，应用$k = 5$，该人的信用度为“low”(低信用度)。



***k*近邻模型中的测试数据集**

我们刚才使用了两次`knn()`函数，虽然我们都称为“测试”但是意义却不同。第一种意义上的“测试”，主要是用来衡量模型效果的。因为在第一个测试集上，我们其实是知道真实的标签值的，我们只是在把测试数据集指定给R的时候假装不知道而已。而第二种测试数据集（比如那个新人的数据）是真正意思上的“样本外测试集”，也就是说，我们真的不知道该人的信用等级如何，而想利用k近邻模型预测得知。可以看到，对于R来说，这两种意义上的“测试”从代码形式上来看是完全一样的，R并不知道你所要做的“测试”是何种意义上的测试。

模型有哪些假设

上一章我们讨论了模型和模型的假设问题。那么对于k近邻模型来说，有哪些模型假设呢？

k近邻从模型形式上来看是一个非参数（nonparamatric）的方法，也就是说，对于数据的生成过程和数据的分布没有任何假设，也没有任何需要估计的参数。但即便作为一个非参数的方法，还是有一些隐含的假设，如下。

- 在数据的特征空间中可以定义某种意义的“距离”。

- 训练数据集中的因变量已经做好标签。
- k值的选择，需要你来决定。
- 我们选择和观测的特征变量对于预测因变量的标签值有所帮助，也就是说，这些特征变量与因变量是有联系的。但很可能某些特征变量与因变量是没有联系的。至于这些特征变量的预测效果到底如何，后期的模型评价自然会告诉我们。在建模的过程中，基于模型评价的反馈，你可能会考虑添加某些特征变量，或者剔除某些特征变量。这些都叫作模型调优，好的模型应该经过仔细地调优。但在调整的过程中，也要注意避免过拟合的问题。

线性回归模型和k近邻模型都属于“监督性学习算法”，也就是说，预测变量的观测值x和待预测变量的观测值y都是已知的，学习的目的是找到x产生y的函数。接下来我们要介绍的算法属于“非监督性学习算法”，它适用于y的观测值未知的数据。

3.2.3 k均值算法

之前介绍的算法都属于监督性学习算法，它适用于y的观测值（对于分类问题来说就是待预测变量的标签值）已知的情况。在这个背景下，我们总是希望模型对y的预测越精确越好，这也是在监督性学习中存在模型评价标准的原因。

“非监督性学习算法”的典型代表是聚类分析，在聚类分析中y的真实标签值是未知的。k均值是我们要接触到的第一个“非监督学习算法”。

假设我们有一些用户层面的观测数据，比如Google＋的数据，某些调查数据，医学实验数据或者SAT分数的数据。

每个用户都可以表示成某些特征变量的组合，比如年龄、性别和收入等。假设下面一行就是用户数据的所有特征变量：

```
age （年龄） gender（性别） income（收入） state（所在州） household size（住户人
```

我们的任务是根据用户变量信息的不同把用户分成几组。这个任务也有很多其他的名字，比如分层、分组、聚类等。但是它们都指的是一个意思，那就是根据用户特征的信息把相似的用户组合起来形成类别。

聚类的目的有很多，下面我们举几个例子说明。

- 市场营销中经常需要根据用户的不同特征，提供给用户不同的产品或者服务。比如，打印机公司肯定只想把墨盒的营销广告展示给已经有打印机的用户。
- 也许你手中的模型只适用于某种类型的群体数据，或者你想对不同的群体数据应用不同的模型。
- 统计学中的分层模型就内含了聚类的想法：比如在家庭消费调查数据中，分层建模会根据地理位置信息的不同，对不同的区域采用不同的模型形式。

为了展示为什么聚类分析是有用的，试想一下在建模过程中，我们经常需要对某些特征变量（属性变量）进行离散化或者分块化（分组）。

比如年龄变量可以分块：20～24周岁、25～30周岁，等等。收入变量也同样可以分块，而像“所在城市”或者“所在州”这样的变量因为已经是分类变量，就不需要再做分块了。但是，如果一个分类变量的类别过多，分块仍然是必要的，当然这要取决于你所采用的模型以及样本数据量的大小。比如，“所在州”变量可以再分块“所在区域”变量，该变量中包含较少的类别：东西部、中部，等等。

假设年龄被分成了10组，性别是两组，并且该被分块的变量都已经完成了分块，那么我们可能会得到$10 \times 2 \times 50 \times 10 \times 3 = 30\ 000$种不同的组合。这对于建模来说，已经是很大规模的分组了。

由于我们有5个分组过的变量，试想一下，在一个五维空间中，每个变量都对应一个坐标轴：因此，有性别轴、收入轴等。每个分组都对应相应的坐标轴点。这样做的结果是这些坐标轴交织形成的网将包含每一个可能的属性变量的组合值。

每个数据点都在这个空间中对应坐标轴系统内的某一个组合点。这样看来，30 000个不同的组合实在是过多了。比如，从市场营销的角度来说，没有任何一家公司想指定30 000种不同的营销策略。

也许这个时候，也才会感到无助而尝试寻求算法的帮助了，尤其是在这种情形下，你想在建模之前进行分组，却又不想得到上万个不同的组别。这就是k均值聚类可以做的事情：k就是这个聚类算法最后所得到的类别数。5

> 5k通常是一个小于10的数，对于市场营销来说，10个类别肯定要比30 000个要容易操作。

二维的问题

刚刚的五维问题可能很难在脑海中有画面感，现在我们讨论一个更简单的二维问题。这里假设数据是有关广告点击率的，我们知道变量是两个，其中x是用户被展示的广告数，另外一个变量y是用户点击广告的次数。

图3-9用散点图展示了数据的大致分布形态。

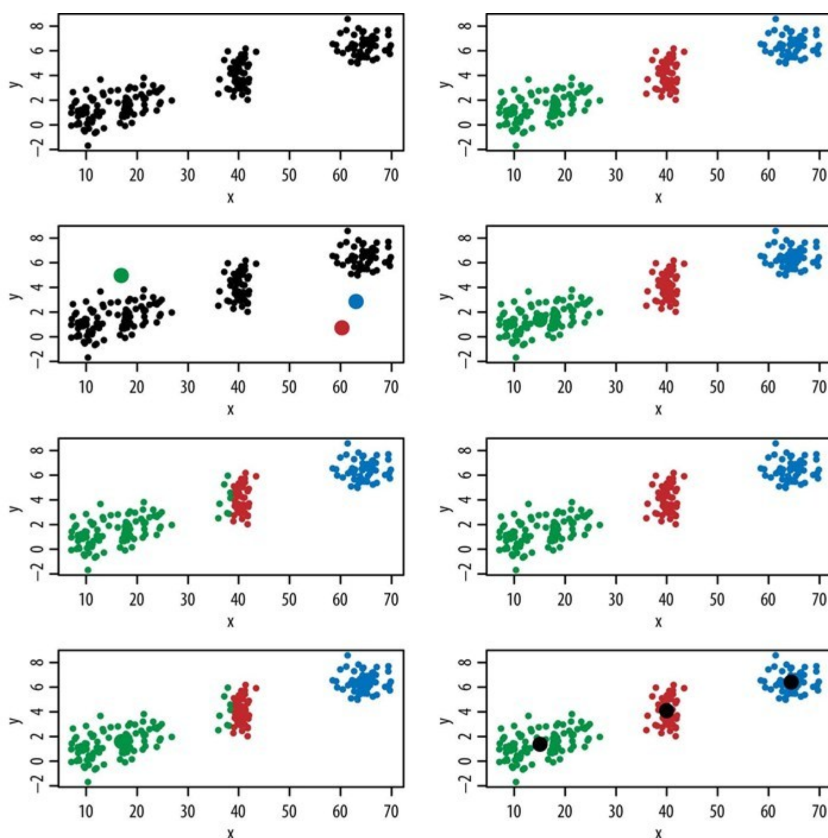


图3-9：二维空间上的聚类过程，先看左半边从上往下，再看右半边从上往下

从图3-9的左上角的散点图可以看出，数据自然地聚成了几类。对于二维空间的数据来说，如果数据点不多，我们很容易就可以通过肉眼判断出数据大概聚成了多少类。当数据维数增加或者数据量变大之后，就不可能依赖于肉眼判断了，你需要一个类似于k均值聚类的算法帮助你判断数据中到底有多少类别。k均值聚类可以在d维空间中找到聚类，其中d是数据空间的维度，也是每个数据点所具有的特征变量的个数。

图3-9展示了聚类的整个过程，可以总结为以下四步。

1. 首先，在数据空间中随机挑选*k*个点（叫作中心点，centroid）。这也叫*k*均值聚类的初始化操作，要尽量让中心点靠近数据点，并且各个中心点之间要明显不同。

2. 将数据点分类到离它最近的中心点。

3. 重新计算中心点的位置。

4. 重复上述两步直到数据中心点的位置不再变动，或者变动幅度很小为止。

至于如何确定算法已经迭代完毕也没有绝对的标准。甚至对于*k*的选择也没有标准，毕竟这是一个非监督性的学习方法，*k*的大小要通过不断地尝试，以确定一个较为合适的值。

这是一个典型的非监督性学习的例子，因为所有的数据点都没有事先做好的标签类别值，都是通过算法来发现。

*k*均值聚类算法有一些已知的缺点。

- *k*的选择是颇具艺术性的。当然，*k*的取值也是有上下限的，需要满足$1 \leq k \leq n$，其中*n*是数据的样本量。
- 收敛性问题：可能不存在唯一解，导致算法在两种可能解之间来回迭代而无法收敛。
- 可解释性问题：也许最终的聚类结果很难给出合理的解释。这通常是*k*均值聚类最棘手的问题。

尽管有很多问题，但是*k*均值聚类算法因为其逻辑简单，运行速度非常快，因此在实际中得到了广泛的应用：包括市场营销、计算机视觉（图像分割）等领域都有较多的应用。*k*均值聚类还可以与其他监督性学习算法结合起来使用，作为某些模型的起始估计值。

在R中，*k*均值聚类同样可以用一行代码搞定：

```
kmeans(x, centers, iter.max = 10, nstart = 1,
         algorithm = c("Hartigan-Wong", "Lloyd", "Forgy",
                       "MacQueen"))
```

代码中的*x*是数据矩阵，矩阵中的每一列代表一个特征变量。最大的迭代次数由`iter.max`所确定，这里我们设定为10，当然你也可以改

为自己认为合理的数值。有很多具体的算法可以实现*k*均值聚类，你可以指定使用哪一个算法。

关于**k**均值聚类的背景知识

我们刚才不是已经详细地解释了这个算法的步骤了吗？它看起来十分简单嘛。其实不然，*k*均值聚类其实经过了很长时间的发展。

1957年Hugo Steingaus和Stuart Lloy各自独立地开发了这个算法的早期标准版本，当然还不叫*k*均值聚类。来自贝尔实验室的James MacQueen在1967年首次使用了这一名称。但直到1982年James关于该名称的论文才得到公开发表（之前是作为贝尔实验室的内部研究而没有对外发表）。

在20世纪60年代和70年代，*k*均值聚类算法得到了很多改进，包括Hartigan-Wong、Lloyd和Forgy等学者都有所贡献。我们之前描述的算法步骤就来自于Hartigan-Wong开发的版本。

直到最近，还会有关于*k*均值聚类算法的新研究发表。其中值得提及的是David Arthur和Sergei Vassilvitskii（现任职于谷歌）于2007年发表的*k*均值＋＋算法。该算法通过最优化初始随机种子有效地避免了算法不收敛的问题。

3.3　练习：机器学习算法基础

继续我们上一章关于纽约市（曼哈顿）房地产的数据分析。相关信息可见http://abt.cm/1g3A12P。

- 应用线性回归模型，分析房地产销售额。选择你觉得可能有用的预测变量，并且验证为什么线性回归模型适用于该数据。
- 可视化线性回归模型的参数和拟合情况。
- 用*k*近邻模做预测。确保你在训练模型之前已经抽离了一部分数据用作独立测试。找到相关的预测变量以及最优的*k*值以达到最小的预测误差。
- 将你的模型结论通过可视化和报告的形式展示给大家。
- 基于模型的分析结果，提出一些可行的建议。

答案

上一章我们讲述了如何清理和探索该数据集。如果你之前没有动手清理过该数据，现在真的是需要动手的时候了，因为在应用回归模型分析数据之前，数据必须是干净的。接下来的两段R代码，第一段是有关于如何针对该数据构建一个线性回归模型的，第二段是有关如何清理和准备数据以应用k近邻模型进行预测的。

示例R代码：房地产数据的线性回归模型

```
作者：Ben Reddy

model1 <- lm(log(sale.price.n) ~ log(gross.sqft),data=bk.homes)
## 想一想这里的代码是用来做什么的

bk.homes[which(bk.homes$gross.sqft==0),]

bk.homes <- bk.homes[which(bk.homes$gross.sqft>0 &
bk.homes$land.sqft>0),]
model1 <- lm(log(sale.price.n) ~ log(gross.sqft),data=bk.homes)
summary(model1)

plot(log(bk.homes$gross.sqft),log(bk.homes$sale.price.n))
abline(model1,col="red",lwd=2)
plot(resid(model1))

model2 <- lm(log(sale.price.n) ~ log(gross.sqft) +
log(land.sqft) + factor(neighborhood),data=bk.homes)
summary(model2)
plot(resid(model2))

## 为了更好的可解释模型，这里去掉模型中的截距项
model2a <- lm(log(sale.price.n) ~ 0 + log(gross.sqft) +
log(land.sqft) + factor(neighborhood),data=bk.homes)
summary(model2a)
plot(resid(model2a))

## 加入building type（建筑种类）变量
model3 <- lm(log(sale.price.n) ~ log(gross.sqft) +
log(land.sqft) + factor(neighborhood) +
factor(building.class.category),data=bk.homes)
summary(model3)
plot(resid(model3))

## 加入neighborhood（邻居个数）和building type（建筑）之间的交叉效应变量
model4 <- lm(log(sale.price.n) ~ log(gross.sqft) +
log(land.sqft) + factor(neighborhood)*
factor(building.class.category),data=bk.homes)
summary(model4)
plot(resid(model4))
```

示例R代码：房地产数据的*k*近邻模型

```
作者：Ben Reddy
require(gdata)
require(geoPlot)
require(class)

setwd("~/Documents/Teaching/Stat 4242 Fall 2012/Homework 2")

mt <- read.xls("rollingsales_manhattan.xls",
pattern="BOROUGH",stringsAsFactors=FALSE)
head(mt)
summary(mt)

names(mt) <- tolower(names(mt))

mt$sale.price.n <- as.numeric(gsub("[^[:digit:]]","",
mt$sale.price))
sum(is.na(mt$sale.price.n))
sum(mt$sale.price.n==0)

names(mt) <- tolower(names(mt))

## 利用正则表达式清理和格式化数据
mt$gross.sqft <- as.numeric(gsub("[^[:digit:]]","",
mt$gross.square.feet))
mt$land.sqft <- as.numeric(gsub("[^[:digit:]]","",
mt$land.square.feet))

mt$sale.date <- as.Date(mt$sale.date)
mt$year.built <- as.numeric(as.character(mt$year.built))
mt$zip.code <- as.character(mt$zip.code)

## 使数据集更加标准化（设定房屋建筑的起始年份为0、土地面积为平方英尺、售价为0的或者负值
min_price <- 10000
mt <- mt[which(mt$sale.price.n>=min_price),]

n_obs <- dim(mt)[1]

mt$address.noapt <- gsub("[,][[:print:]]*","",
gsub("[ ]+"," ",trim(mt$address)))

mt_add <- unique(data.frame(mt$address.noapt,mt$zip.code,
stringsAsFactors=FALSE))
names(mt_add) <- c("address.noapt","zip.code")
mt_add <- mt_add[order(mt_add$address.noapt),]

# 找重复值并删除，这些重复值往往是一个地址，但邮编却不同
dup <- duplicated(mt_add$address.noapt)
# 删除这些重复值
dup_add <- mt_add[mt_add$dup,1]
mt_add <- mt_add[(mt_add$address.noapt != dup_add[1] &
mt_add$address.noapt != dup_add[2]),]
```

```
n_add <- dim(mt_add)[1]

# 随机选取500个地址，不然程序永远跑不完
n_sample <- 500
add_sample <- mt_add[sample.int(n_add,size=n_sample),]

# 首先用手头的一个地址尝试一下
query_list <- addrListLookup(data.frame(1:n_sample,
        add_sample$address.noapt,rep("NEW YORK",times=n_sample),
        rep("NY",times=n_sample),add_sample$zip.code,
        rep("US",times=n_sample)))[,1:4]

query_list$matched <- (query_list$latitude != 0)

unmatched_inds <- which(!query_list$matched)
unmatched <- length(unmatched_inds)

# 尝试把EAST/WEST改作E/W
query_list[unmatched_inds,1:4] <- addrListLookup
      (data.frame(1:unmatched,gsub(" WEST "," W ",
      gsub(" EAST "," E ",add_sample[unmatched_inds,1])),
     rep("NEW YORK",times=unmatched), rep("NY",times=unmatched),
            add_sample[unmatched_inds,2],rep("US",times=unmatched)))[,1:4]

query_list$matched <- (query_list$latitude != 0)
unmatched_inds <- which(!query_list$matched)
unmatched <- length(unmatched_inds)

# 尝试把 STREET/AVENUE 改作 ST/AVE
query_list[unmatched_inds,1:4] <- addrListLookup
        (data.frame(1:unmatched,gsub(" WEST "," W ",
        gsub(" EAST "," E ",gsub(" STREET"," ST",
        gsub(" AVENUE"," AVE",add_sample[unmatched_inds,1])))),
        rep("NEW YORK",times=unmatched), rep("NY",times=unmatched),
            add_sample[unmatched_inds,2],rep("US",times=unmatched)))[,1:4]

query_list$matched <- (query_list$latitude != 0)
unmatched_inds <- which(!query_list$matched)
unmatched <- length(unmatched_inds)

## 现在我们已经做得差不多了
add_sample <- cbind(add_sample,query_list$latitude,
query_list$longitude)
names(add_sample)[3:4] <- c("latitude","longitude")

add_sample <- add_sample[add_sample$latitude!=0,]

add_use <- merge(mt,add_sample)

add_use <- add_use[!is.na(add_use$latitude),]

# 地理坐标值
map_coords <- add_use[,c(2,4,26,27)]
```

```
table(map_coords$neighborhood)
map_coords$neighborhood <- as.factor(map_coords$neighborhood)

geoPlot(map_coords,zoom=12,color=map_coords$neighborhood)

## - 使用knn函数
## - 当然可以用的方法还有很多，此处不讨论过多


map_coords$class <- as.numeric(map_coords$neighborhood)
n_cases <- dim(map_coords)[1]
split <- 0.8

train_inds <- sample.int(n_cases,floor(split*n_cases))
test_inds <- (1:n_cases)[-train_inds]

k_max <- 10
knn_pred <- matrix(NA,ncol=k_max,nrow=length(test_inds))
knn_test_error <- rep(NA,times=k_max)

for (i in 1:k_max) {
        knn_pred[,i] <- knn(map_coords[train_inds,3:4],
        map_coords[test_inds,3:4],cl=map_coords[train_inds,5],k=i)
        knn_test_error[i] <- sum(knn_pred[,i]!=
        map_coords[test_inds,5])/length(test_inds)
}

plot(1:k_max,knn_test_error)
```

大规模数据建模及算法

迄今为止我们分析过的数据都难称为大数据。如果遇到大型的数据，我们的模型和算法会发生怎样的改变呢？

在大多数情况下，大型数据都可以拆分为一系列小数据集之后再分别独立建模（甚至可以使用同一个模型）。这通常叫作大数据的“碎片化”（sharding，碎片化就是指将大数据分割成很多小的部分并分配给不同的计算机进行建模，模型的参数估计会以分布的形式返回给建模者）。

碎片化固然是大规模数据建模的一个较为简单和直接的解决方案。然而，当数据的规模达到一定的级别，模型本身就需要优化以解决计算量过大的问题。比如，线性回归模型的参数估计过程严重依赖于矩阵的求逆操作。对于大数据来说，矩阵的求逆操作是不现

实的，因此应该考虑如何近似化矩阵的求逆以减轻计算负担，从而使得线性回归可以应用于分析大数据。

基于大数据分析需求的模型优化是大数据领域的研究前沿，它需要技术和理论的同步发展。Edinburugh大学的Peter Richtarik于2013年有过一段关于大数据下模型优化的演讲，他说道："在大数据领域，传统的基于最优化方法的模型，因为涉及大量的迭代操作，给计算造成了巨大的负担，有时候即便是一次迭代也很难在短时间内完成。因为大多数统计模型在开发之初都没有考虑过大数据的适用问题，就算在10年前我们也很难预见数据会达到如今的规模。因此，我们亟需一些简单的、数据友好的、内存要求低的的模型和算法，以适应大规模数据分析。解决平行计算框架问题（包括多核计算问题、GPU计算问题以及计算机集成技术）才能有效解决大数据分析的海量计算问题。"

对该问题的讨论已经超出了本书的范畴，但是作为一名数据科学家，我们要知晓这些问题的存在。在第14章我们还会对其中的某些细节问题做进一步的讨论。

3.4 总结

本章介绍了机器学习领域三个基础算法，它们已经被广泛地应用于数据分析的各个领域。如果你已经理解和掌握了这三个算法，那么对于数据科学来说，你已经基本入门了。如果你还是觉得理解和掌握这些基本的算法有点困难，不用灰心丧气，这本来就不是一段轻松的学习旅程。

在很多情况下，对于预测和分类问题来说，线性回归模型都是最基础的模型。本章已经为大家展示了如何应用线性回归模型预测一个连续型的数值型变量，模型中可以使用多个预测变量。在第5章，我们还会更详细地讨论线性回归模型，我们还会学习到逻辑回归模型，它可以用来预测二元分类变量；第6章将讨论时间序列模型。在第7章，我们会为大家介绍有关模型中特征变量的选择问题。

k近邻和k均值聚类是两种聚类算法，适用于把相似的事物归类的问题。对于聚类算法来说，最重要的就是距离的定义和模型的评价标准

这两个问题。它们的解决方式都具有一定的灵活性和主观性。下一章我们会继续讨论有关聚类算法的问题，其中朴素贝叶斯算法将是下一章的重点。在第10章我们会聊到关于社交网络数据的分析问题，社交网络分析中的图聚类模型是非常有意思的研究领域。本书没有涉及的聚类算法有分层聚类和基于模型的聚类，有兴趣的读者可以自己找资料了解一下。

对于聚类分析感兴趣的读者，我们推荐Hastie和Tibshirani的著作*Elements of Statistical Learning*（《统计学习基础》，http://stanford.io/16hcTKn）一书，英文版由Springer出版社出版。想要深入理解贝叶斯视角下的回归模型，我们极力推荐Andrew Gelman和Jennifer Hill的著作*Data Analysis using Regression and Multilevel/Hierarchical Models*（《基于回归和多层/分层模型的数据分析》）。

3.5 思维实验：关于统计学家的自动化

Rachel在2013年5月参加了在伦敦帝国理工学院（Imperial College London）举办的大数据挖掘会议。来自于剑桥大学的Zoubin Ghahramani教授是会议的嘉宾之一，他在其中的一个研讨会上提到了“自动化统计学家”的概念，他说打造一个“自动化统计学家”是他计划要完成的长期项目之一。对于“自动化统计学家”这个概念你有什么想法呢？如果让你打造一个“自动化统计学家”你该如何着手呢？

第 4 章　垃圾邮件过滤器、朴素贝叶斯与数据清理

本章的贡献主要来自于Jake Hofman先生。Jake不久前刚从雅虎研究院离职，随后加入了微软研究院。他曾在哥伦比亚大学主修物理学，并获得博士学位。毕业之后，他留任哥伦比亚大学并长期教授一门广受学生欢迎的数据建模课程。近期他也开始教授一门新课，课程名叫作“社会科学中的计算方法”。

让我们先来了解一下Jake的数据科学背景。他本人曾经给数据科学添加过一项必备技能，名曰“数据清理”。他说自己不知道为什么在数据清理上花了那么多的时间，然而，熟能生巧，他俨然已经成为了数据清理方面的权威专家。

4.1　思维实验：从实例中学习

首先看一下图4-1，这看起来应该是某人电子邮箱首页的截图，图中每一行显示的是一封邮件的概要。

再仔细看一下，你可能已经发现某些邮件乍一看就能判断出是垃圾邮件。

但是现实的问题是，我们如何才能确认一封邮件是否是垃圾邮件呢？我们的大脑可以综合各方面信息对一封邮件的性质做出逻辑推断，那么你能否通过编程来模拟人脑识别垃圾邮件的过程呢？

Pure Saffron Extract	Melt Fat Away - Drop 11-lbs in 7 Days! - Melt Fat Away - Drop 11-lbs in 7 Days! Melt Fat Away - Drop 11-lbs i
Blue Sky Auto	Car Loans Available - Bad Credit Accepted
Watch The Video	Shocking Discovery Gets You Laid - Scientists at Harvad University have discovered a strange secret that allo
Casino	Casino Promotions - With the Slots of Vegas Instant-Win Scratch Ticket Game you can get $100 on the hous
Designer Watch Replica	Replica Watches On Sale - Replica Watches: Swiss Luxury Watch Replicas, Rolex, Omega, Breitling Check
A.C., me (10)	I'm late to this party - I'm free and interested. Tell me more! I'd have to think about the students, but I know so
Rachel .. Christoforos (18)	Fwd: Invitation to speak at upcoming Big Data Workshop, hosted by Imperial College London - Dear Rachel, th
Fat Burning Hormone	17 Foods that GET RID of stomach fat
Kaplan University	Kaplan University online and campus degree programs
Dinn Trophy	Sport Plaques - As Low As $4.29 - View this message in a browser. Shop Sport Plaques Shop Now> Change
me, Philipp (2)	checking in - Hi Rachel, I know! I had started writing a few emails to you, but then I (obviously) didn't sent
me, Matthew (3)	doing data science - Hi Matt, Not a duplicate (just FYI if that helps debug) Well, so the status is that we're in t
Luxury Replicas	Rolex, Breitling, Chanel, Omega, LV, and muchMore! - Super Replicas - Luxury Watches, Bags, Jewelry, Pho
Watch this video and wom.	Watch this video and women will adore you - Can you get laid using just the words in this video? Click Here To
Adriana	I ADDED YOU to my Private Wish List - Sorry, I've been out of town but I am back and I'm looking for a good ti

图4-1：邮箱中的垃圾邮件

Rachel的课程给了我们一些启示：垃圾邮件通会与某些特征指标紧密相关。

- 如果邮件中包含任何有关“伟哥”的信息，那么基本上可以确定是垃圾邮件。这是一条非常明晰的规则，但也许垃圾邮件的发送者也同样了解该规则， 并可以通过改变拼写等方法成功地规避检查（其实，一些垃圾邮件制造者聪颖过人，只是他们没有把智慧用在该用的地方）。
- 邮件主题的字符长度，感叹号（或者其他标点符号）的使用频率等指标也可以作为垃圾邮件的特征指标。但也有例外：譬如雅虎公司的商标名是“Yahoo!”，其中的感叹号是该商标的设计元素，因此不能被一视同仁地视作垃圾邮件的标识。也就是说，垃圾邮件的特征指标设计也不能过于简单和严苛。

如果你想通过编程实现垃圾邮件的过滤，下面几点是我们的建议。

- 尝试使用概率模型。换句话说，避免使用过于简单的过滤指标。更好的办法是通过指标组合的方式，利用概率模型计算邮件是垃圾邮件的概率。在我们看来，概率模型非常适合用来搭建垃圾邮件过滤器。
- 在前一章我们已经学习了*k*近邻以及线性回归模型，那么这两个模型可否用在此处呢？（答案是：不可以。大家可以先想一想为什么。）

本章我们将详细讨论如何应用朴素贝叶斯模型搭建一个垃圾邮件过滤

器。 理论上看来，朴素贝叶斯是一个介于k近邻和线性回归模型之间的方法。至于原因，此处说来话长，让我们先从线性回归说起。

4.1.1 线性回归为何不适用

线性回归大家都不陌生了，它是我们工具箱中的必备神器。首先我们讨论一下该模型的适用条件以及它为何不适用于垃圾邮件过滤问题。为了便于建模，我们将邮件数据先转换成数据矩阵，该矩阵的每一行代表某封邮件的信息（可以用`email_id`来标明邮件号），邮件中每个出现过的单词被称作“特征”，它们被置于矩阵的列上。举例来说，单词“伟哥”可以构成了矩阵的某一列，如果某封邮件出现了至少一次“伟哥”，那么在该邮件行的“伟哥”这一列上填上数字1，代表“伟哥”在该封邮件中至少出现了一次；否则填上数字0。当然，我们也可以在这个位置填上“伟哥”出现的实际次数。至于到底怎么填，要由数据分析人员事先确定。

回想一下前一章有关线性回归的内容，在建模之前我们首先需要一个关于邮件详情的训练数据集。在这个数据集中，每一封邮件已经被标明了是否是垃圾邮件。那么一个自然的问题就是，如何确定一封邮件是否是垃圾邮件呢？一种做法是人工核查每封邮件，并为每一封邮件打上是否为垃圾邮件的标签信息。该方法固然可行，但会耗费大量的人力物力财力。另一种做法是应用市面上已有的垃圾邮件过滤器为每一封邮件打上标签信息，比如使用Gmail声名远扬的垃圾邮件过滤系统。（此处一个自然的问题是，如果已经有了像Gmail这样一个业已成熟的垃圾邮件处理器，为啥我们还非要自己动手做一个呢？这是个好问题，不过在这里我们只是想通过演示如何搭建一个类似Gmail的过滤器去学习相关的算法）。一旦搭建好了模型，我们就可以用它预测一封没有标签的新邮件到底是否为垃圾邮件。**1**

> 1此处的意思是，训练数据集必须是已经做好标签的邮件数据，这样模型也有Y变量。

不适用线性回归的首要原因是此处的目标Y变量（邮件是否为垃圾邮件）是一个二元变量（0代表垃圾邮件，1代表正常邮件）。我们知道，线性回归的预测值是一个连续性的变量，它可能是实数域上的任何一个数值，因此它的预测值不可能只是一个二元值。所以从模型输出值的性质来说，线性回归不适用于二元变量的建模，而更适用于连续性变量。

基本上可以说，线性回归对于垃圾邮件的识别问题是束手无策的，我们应该想办法找到一个更适用于二元变量的模型。但其实，如果我们

生搬硬套，仍然尝试在R中使用线性回归模型搭建邮件过滤器，从理论上来说我们还是会得到回归模型的所有典型的输出结果。因为R软件本身不会告诉我们某个模型是否合适。我们还是可以用线性回归，用R估计出模型参数以及预测值。因为此时的预测值是一个连续性的数值，为了得到二元预测值（0/1）的输出结果，我们需要设定一个阈值，并且规定：如果预测值在阈值之上就设定预测值为1，否则为0。

即便如此生搬硬套，对于这样一个数据集来说，线性回归模型仍然会失败。为什么呢？因为对于邮件数据来说，变量的个数相比样本量的个数实在太大了。对于一个典型的邮件数据集，样本量的单位级在万左右（假设有1万封邮件），而其中所有可能的单词个数可能多达10万个，也就是说，变量个数为10万。传统的线性回归不能处理这种变量个数多于样本量的情形。线性回归模型参数估计的理论告诉我们，这会导致线性回归最小二乘解中的矩阵不可逆。就算换个角度，从计算机存储的角度来说，处理这样规模的数据集，普通的个人计算机还是会显得捉襟见肘。

接下来很自然会想到，也许我们可以从10万个单词出挑选出最经常出现的1万个单词，这样最起码我们可以得到可逆的估计矩阵了。但即便如此，模型的估计结果也会非常不稳定。为了彻底解决变量过多的问题，假设我们可以把单词的变量个数缩减到100个（比如，邀请垃圾邮件领域的专家仔细挑选最有用的单词变量），那么我们就顺利解决了模型的估计问题了。真的如此吗？其实不然，即便我们折腾到现在，还是没有解决一个最初也是最根本的问题：线性回归根本不能预测输出二元变量值！



番外话：垃圾邮件过滤器之现状

过去的5年中，研究人员开始使用随机梯度的方法解决我们刚才提到的模型估计矩阵不可逆的问题。例如，随机梯度方法可以和逻辑回归模型结合用来预测二元目标变量，而且这种方法能够考虑到单词变量之间的相关性。然而，在垃圾邮件过滤问题上，朴素贝叶斯模型显示了强大的适用性，它足够简单，效果也足够好。

4.1.2 *k*近邻效果如何

听起来朴素贝叶斯模型十分诱人。别着急，我们很快就会讲到它。在这之前不妨让我们先回顾一下之前学习过的k近邻模型。对于任何模型来说，首要的任务还是要选择一些特征变量。沿用之前的做法，我们仍然使用单词作为特征变量。如果一个单词出现在邮件中，相应的变量值为1，否则为0。对于k近邻模型来说，最重要的莫过于确定“近邻”的含义。在什么样的情况下，两封邮件才可谓“近邻”呢？或者说，两个给定特征变量的邮件到底有多“近邻”呢？一个最直接的想法就是，当这两封邮件包含的单词十分相似时可将它们视为“近邻”。

之前的变量个数大于样本量的问题在这里同样困扰着我们，虽然在k近邻方法中我们不会遇到计算逆矩阵的问题，但是我们还是有一个大麻烦：数据空间的维度太高了。我们有1万封邮件和10万个单词，这就代表数据的空间高达10万维。从计算量上来说，在如此高维度的空间里计算相似度是很难的。

但其实计算量也不算是个大问题，最主要的问题是，在如此高维度的空间内，即便是两个最近的“邻居”也会相距深远。这通常称作“高维诅咒”，它会严重影响k近邻的模型效果。

番外话：数字识别

图4-2是10行从0～9的手写数字，假设我们想用一种算法识别这些手写数字，那么k近邻会是个不错的选择。

图4-2：手写数字

要想建模，首先需要处理手写图像数据，将其转换成可以分析和建模的形式。譬如说，我们可以用一个16×16的像素框表示一个手写数字，框内的每个像素的亮度值代表了这个数字在该像素上的表示强度。

然后再将这个16×16的像素框拉直成一个长度为256的向量。这样做的目的是便于我们接下来应用阿基米德测度计算距离。换句话说，我们成功地将每一个手写数字转化成了一个长度为256的向量。这样一来，两个手写数字的差异程度则可以用两个向量的阿基米德距离来表示。向量的阿基米德距离就是向量所有元素的“离差平方和”。在空间上解释就是两个向量间的点对点长度。完成上述的形式转化和距离定义之后，我们就可以着手建模了。

k近邻的模型效果会随着近邻个数的改变而改变。我们需要调整好k的大小以尽量避免类似过拟合的现象出现。如果你已经认真做完了形式转化和距离定义的工作，并仔细地挑选了一个合理的k值，那么k近邻的模型效果通常都不会让你太失望。即使是将算法扩展到一个较大的数据集，模型准确度也通常会达到97%。

模型最后的预测效果通常用一个“混淆矩阵”表示。假设模型的任务是将识别结果分到可能的k个类别中（在这个手写数字识别的例子中，$k = 10$），那么混淆矩阵就是一个$k \times k$的矩阵。该矩阵中的列代表实际的标签值，行代表了模型预测的标签值。因此该矩阵的第(i, j)个元素就代表有多少个结果，其实际的标签是i却被模型预测成了标签j。给定一个混淆矩阵就可以轻松地得到模型的预测准确度：也就是所有被正确预测的标签数占总标签数的比例。在前一章中，我们介绍了误分率的概念，而这里的预测准确度 = 1 - 误分率。

4.2　朴素贝叶斯模型

我们似乎走投无路了，因为线性回归和k近邻两大神器在垃圾邮件过滤问题上似乎都不太给力。不要灰心！接下来我们就立马介绍朴素贝叶斯模型，对于垃圾邮件过滤问题，它可以快刀斩乱麻。

4.2.1　贝叶斯法则

让我们通过一个更为简单的例子感受一下为什么朴素贝叶斯会大有作为。设想我们在检测一种罕见的疾病，其在人群中的发病率只有

1%。我们的测试方法非常适用于此病，具有很高的精度，但也并非100%的完美。其表现为：

- 99%的病人可以被检测出患有此病（测试结果为阳性）2；
- 99%的健康人可以被检测出未患此病（测试结果为阴性）3。

> 2这也叫作“真阳性”。

> 3这也叫作“真阴性”。

那么现在的问题是，如果一个人的测试结果为阳性，那么他真正患有此病的概率是多少呢？

让我们先用一个笨方法回答这个问题。假设有$100 \times 100 = 10\ 000$个群众参与了测试，那么根据发病率，这个人群中会有100个人患有此病，而剩下的9900个人是健康的。该测试的结果是100个患病人口中会有99个被检测为患有此病；然而同样的，9900个健康人口中同样也会有99个会被检测为患有此病。也就是说，如果一个人的测试结果为阳性，那么他真实的患病概率是对半开，50%！图4-3中的树形逻辑图更好的刻画了我们刚才的推理过程。

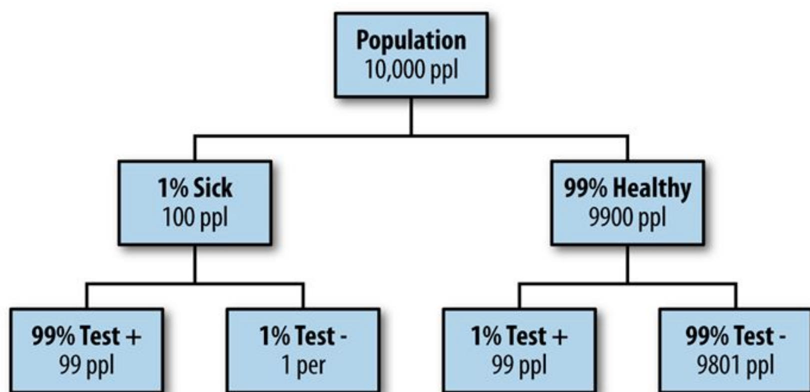


图4-3：树形逻辑推断图

我们不如把刚才的推理过程用公式表示一下，这样会显得我们比较聪明。回想一下你可能上过的初等统计学的课程，给定两个事件x和y，其各自发生的概率分别为$p(x)$和$p(y)$。它们联合发生的概率（表示为$p(x,y)$）以及它们相互发生的条件概率（比如说$p(y|x)$就表示给定事件x发生的情况下，事件y发生的概率）有如下关系：

$$p(y|x)p(x) = p(x,y) = p(x|y)p(y)$$

应用此式，我们可以得到贝叶斯法则并进而得到关于$p(y|x)$的概率表示（此处假设$p(x) \neq 0$）：

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)}$$

其中的分母项$p(x)$可视作一个“正则常数”，并且通常来说得到它的值相对比较容易。结合之前的例子，这里y对应于事件“患有此病”，此处用Sick表示；x对应于“测试结果为阳性”，此处用+表示。应用上述的贝叶斯公式就可以计算出在给定测试结果为阳性的条件下某人真实患病的概率：

Invalid Equation

结果与我们使用笨方法得出的结论完全一致！

4.2.2　个别单词的过滤器

准备工作已经做得比较到位了。我们应该捋起袖子，想一想到底如何应用如此简单的贝叶斯法则搭建一个靠谱的垃圾邮件过滤器呢？的确，如果一封邮件中有“伟哥”这样的单词，那么它极可能就是一封垃圾邮件。但是这压根还没完啊，我们还需要考虑邮件中可能出现的形形色色的各种单词呢！

不如让我们把每个单词抽象出来，各个击毙。这里，单词用“word”表示，垃圾邮件用“spam”表示。应用贝叶斯公式我们可以计算出，如果一个单词出现，该邮件可能是垃圾邮件的概率：

$$p(\text{spam}|\text{word}) = \frac{p(\text{word}|\text{spam})}{p(\text{word})}$$

如果有足够多的已经做好标签的训练数据，这个等式右边的诸项都很容易计算出来。具体来说，如果用“ham”表示正常邮件，那么我们只需计算几个概率值：$p(\text{word}|\text{spam})$、$p(\text{word}|\text{ham})$、$p(\text{spam})$以及$p(\text{ham}) = 1 - p(\text{spam})$。等式右项中的分母部分我们已经知道怎么算了（如果不会，看下之前医学检测的例子）：

$$p(\text{word}) = p(\text{word}|\text{spam})p(\text{spam}) + p(\text{word}|\text{ham})p(\text{ham})$$

可以说我们已经把问题简化成了一个计数的问题：通过计算所有邮件中垃圾邮件的比例就可以得到概率p(spam)，然后在所有的垃圾邮件中计算某一个特定单词出现的频率，我们又可以得到概率p(word|spam)。更进一步，在所有的正常邮件中计算一个特定单词出现的频率我们又轻松地得到了概率p(word|ham)。

大功告成！贝叶斯公式加上简单的计数工作解决了所有的问题，接下来就是你自己动手的时候了。上网把Enron公司的电子邮件数据下载到你的个人电脑上（https://www.cs.cmu.edu/~enron/），接着就开始动手在这个数据上搭建一个垃圾邮件过滤器吧！这将意味着我们会替Enron公司的雇员们搭建一个全新的垃圾过滤系统，这个系统比他们正在使用的现存过滤系统要有效得多。当然，我们会根据Enron公司对于垃圾邮件的定义来设定关键词，使得新的过滤系统更适合该公司的需求。（这也意味着，如果从2001年开始，垃圾邮件的制造者们也从这些数据中学到了一些规则，他们就会在发送垃圾邮件的时候尽力规避，那么我们搭建的过滤器的功效可能就要大打折扣了。）

我们可以用bash写一段shell脚本实现这个功能，记得Jake之前就是这么干的。该段代码的功能是从网上下载垃圾邮件数据集并解压缩到一个新的文件夹中。每一封邮件都对应一个文本文档，并且垃圾邮件和正常邮件应该置于不同的文件夹中以便区分。

这个邮件数据库中的某些统计量指标似乎唾手可得。比如，我们可以数出来总共有1500封垃圾邮件和3672封正常邮件，因此p(spam)和p(ham)的概率值已经在我们手中了。使用命令行工具，我们可以进一步在垃圾邮件文件中计算出单词“meeting”出现的次数：

```
grep -il meeting enron1/spam/*.txt | wc -l
```

我们得到的数值是16。在正常邮件文件夹中重复上述操作，得到了数值153。有了这些信息便可以轻易地计算出给定一封邮件中出现过单词“meeting”，它可能是垃圾邮件的概率：

$$\hat{p}(spam) = 1500/(1500+3672) = 0.29$$
$$\hat{p}(ham) = 1 - 0.29 = 0.71$$
$$\hat{p}(meeting|spam) = 16/1500 = 0.0106$$
$$\hat{p}(meeting|ham) = 153/1500 = 0.0416$$
$$\hat{p}(spam|meeting) = \frac{\hat{p}(meeting|spam)\hat{p}(spam)}{\hat{p}(meeting)} = \frac{0.0106 \cdot 0.29}{(0.0106 \cdot 0.29 + 0.0416 \cdot 0.71)} = 9\%$$

很明显我们根本不需要一个复杂的程序去实现上述简单的计算操作。

接下来让我们尝试一些别的单词，应用贝叶斯法则可以计算出如果一封邮件中出现了某个单词，该邮件可能是垃圾邮件的概率。下面是一些计算结果，看起来似乎效果不错。

- money（金钱）：80%
- vigra（伟哥）：100%
- enron（安然公司名）：0%

现在模型已经工作了，但其实细想想，因为我们使用了Enron公司关于垃圾邮件的定义，模型很容易过拟合。也就是说，对于该模型我们不能过分自信，它很可能只能用在Enron公司的数据上。比如说，我们能否拍胸脯说，如果一封邮件中主要出现“伟哥”这一单词就铁定是一封垃圾邮件呢？肯定不能，因为从常理上来说，我们很容易写一封正常的邮件并在里面开玩笑的写上“伟哥”这一单词。同样的道理，我也可以编造一封垃圾邮件，但只要在其中添加“enron”一词，它便可以轻松通过我们刚刚搭建的过滤器的检查了。

4.2.3　直通朴素贝叶斯

现在让我们把所有单词的信息都利用起来，搭建一个真正的朴素贝叶斯模型。每封邮件都可以表示为一个二元向量，这个向量的第j个元素是0还是1取决于第j个单词是否出现在这封邮件中（出现为1，否则为0）。向量的长度取决于总共要考虑的单词个数。如果要考虑所有在邮件中出现过的单词，那么这个向量必然会很长。

此时模型的输出值就是在给定一份邮件的标签值之后（也即知道它是否是垃圾邮件之后），这封邮件所代表的向量中的单词一起出现的概率。用x表示一封邮件的单词向量，x_j表示向量中的某个元素，下标j代表某个单词在向量中的位置。c代表垃圾邮件，那么上句中的概率值为：

$$p(x|c) = \prod_j \theta_{jc}^{x_j} (1 - \theta_{jc})^{(1-x_j)}$$

式中的θ代表某个单词在垃圾邮件中出现的概率。上一节我们已经阐述了如何用计数的方法得到这个概率，此处我们假设每个单词在垃圾邮件中出现的概率都已经计算出来了，因此视为一个已知量。

朴素贝叶斯模型的核心概念是独立性，这也是我们能够在上式的右边使用连乘符号的原因：因为我们假设单词之间出现与否是相互独立的。这个独立性假设也正是朴素贝叶斯方法中“朴素”一词的含义。这个假设在现实中是很难成立的，比如某些单词会倾向于成双结对地出

现。因此在做独立性假设的同时，我们便忽略了这种可能同时出现的情形。4

> 4当然，独立性假设还排除了很多别的情形，并不仅限于“单词同时出现”的情况。

回到刚才的等式中，对于概率连乘我们通常会在等式两边取对数，将连乘转变成连加的形式：

$$\log(p(x|c)) = \sum_j x_j \log(\frac{\theta_j}{1-\theta_j}) + \sum_j \log(1-\theta_j)$$



在连乘项中如果存在很多接近0的数值很可能会导致数值计算的不稳定。取对数恰好也解决了这个问题。

上式中的Invalid Equation与邮件本身没有关系，而只取决与某个单词本身的性质。我们可以把它重命名为w_j。同样假设$\sum_j \log(1-\theta_j) = w_0$，于是上式可简化为：

$$\log(p(x|c)) = \sum_j x_j w_j + w_0$$

式中唯一跟邮件本身有关的就是x_j了，这个也不难计算。

现在我们终于可以集合所有得到的信息计算$p(x|c)$了，然后贝叶斯法则会帮助计算出我们真正想知道的概率值$p(c|x)$：贝叶斯公式中的其他部分的计算，相比$p(x|c)$都较为简单。如果你只关心一封邮件更有可能是垃圾邮件还是正常邮件，那么你甚至不需要计算贝叶斯公式中的其他部分，而只需要计算$p(x|c)$。因为只有它是跟邮件本身有关的变动项。

注意到没，最后的等式跟线性回归模型的等式十分类似。唯一不同的地方在于对于参数w_j的估计我们在这里使用了贝叶斯公式而不是通过计算逆矩阵。

朴素贝叶斯模型的精度很高，并且非常“实惠”。如果已经准备好了一个已经做好标签的数据集，可以立即上马朴素贝叶斯模型，它的训练

非常容易。即便是过滤成千上万封邮件，我们要做的也仅仅是在垃圾邮件和正常邮件中数一数单词们出现的频率。如果有更多的数据，模型的更新也非常地迅捷：只要更新相应单词的频率就可以了。在实际应用中，通常会有一个简单的基础模型以备使用，我们要做的就是在这个模型基础上加以改良并个性化地运用到我们自己手中的数据上。所以，即便市面上有很多花里胡哨的复杂模型，而我们不妨用用朴素贝叶斯这样简单有效的模型。

如果你想更深入地了解贝叶斯法则和朴素贝叶斯模型，以下是一些不错的参考资料：

- “Idiot's Bayes - not so stupid after all”（“给笨蛋的贝叶斯统计学：其实没有那么笨”，整篇文章都在论述为什么贝叶斯是个好方法；参见http://goo.gl/sD86Oj）；
- “Naive Bayes at Forty: The Independence Assumption in Information”（“不惑之年的朴素贝叶斯方法：信息中的独立性假设的价值”，参见http://goo.gl/JqjRg3）；
- “Spam Filtering with Naive Bayes - Which Naive Bayes?”（“垃圾邮件过滤与朴素贝叶斯模型——什么是贝叶斯统计学”，参见http://goo.gl/74CxlZ）。

4.3　拉普拉斯平滑法

还记得刚才的θ_j吗？它代表垃圾邮件中某个单词出现的概率。仔细想想，它无非就是一个商数：$\theta_j n_{jc}/n_c$，其中n_{jc}代表在该垃圾邮件中该单词出现的总次数，而n_c则代表在所有邮件中（包括垃圾邮件和正常邮件）该单词出现的次数。

拉普拉斯平滑法乍看起来相当无厘头：

$$\theta_{jc} = (n_{jc} + \alpha)/(n_c + \beta)$$

只要满足n_{jc}是一个概率值（位于0和1之间），α和β可以任意取值。比如，我们可以设定$\alpha = 1, \quad \beta = 10$，但是其背后的道理却很难一眼就看出来。其实拉普拉斯平滑与刚才讲到的朴素贝叶斯关系相当密切。如果在朴素贝叶斯模型中加入先验信息，并应用最大似然估计法估计参数，拉普拉斯法变得花哨起来了。用ML表示最大似然估计法，数据用*D*表示，那么参数的最大似然估计可以表示为：

$$\theta_{\mathrm{ML}} = argmax_{\theta} p(D|\theta)$$

而刚才我们提到的$\theta_j = n_{jc}/n_c$就是最大似然估计的结果，它的含义是：这样的估计最有可能生成这样的数据D。朴素贝叶斯中最关键的假设条件就是“独立性假设”，如果该假设在这里仍然成立，那么对于每一个j我们都可以分别用一个θ_j最大化其相应的似然值：

$$\log\left(\theta_j^{n_{jc}}(1-\theta_j)^{n_c-n_{jc}}\right)$$

对其取一阶导并令为0，可以得到：

$$\hat{\theta}_j = n_{jc}/n_c$$

这正是我们在朴素贝叶斯模型中得到的结果。这意味着，如果“独立性假设”成立，朴素贝叶斯中θ的估计值恰好就是它的最大似然估计值。

如果加上先验信息，结合似然函数便可以得到后验估计值，进而得到最大后验似然估计（MAP）：

$$\theta_{\mathrm{MAP}} = argmax\, p(\theta|D)$$

这相当于在回答这样一个问题：如果已经观测到了所有数据，哪个θ值是最有可能的？

贝叶斯法则在这里起到了关键性的桥梁作用，它把估计量θ_{MAP}与$p(D|\theta)\cdot p(\theta)$紧密地联系了起来：它们是严格地正比例关系。其中，$p(\theta)$称作“先验信息”，通常我们要人为地赋予其一个合理的概率值。如果我们假设θ服从一个最简单的Beta分布$\theta^{\alpha}(1-\theta)^{\beta}$，那么$\theta$的最大后验估计量$\theta_{\mathrm{MAP}}$就是本节刚开始所展示的拉普拉斯平滑值。

> 假设的先验分布合理吗
>
> θ的含义是：如果邮件中出现某个单词那么该封邮件可能是垃圾邮件的概率。假设它服从一个Beta分布，那么只要α和β都大于0，它的概率分布在0和1两侧的取值都接近于0。我们说过，我们不能100%地确认一个单词的出现就代表这是一封垃圾邮件，因此这样的概率分布假设应该是合乎常理的：在垃圾邮件中，几乎没有单词会绝对出现/不出现，我们不想表现得过于极端。
>
> 然而，如果α和β取比较大的值，那么θ的大部分值都

会集中在其分布的中间部分。也就是说大部分的θ都十分相近。从垃圾邮件过滤的角度来看，大部分的单词都以相似的概率出现在垃圾邮件和正常邮件中，这似乎不是一个合理的假设，因为总有一些单词与垃圾邮件的关系更加密切。

一个折中的方法是把α和β设定为一个很小的正值，比如说0.2。这样我们的模型才既不会过于极端，也不会过于平庸。对于先验信息分布，规矩是，数据量愈大可以愈发放松对它的假设。因为，大样本量代表的似然函数部分的效果会极大地掩盖先验信息的影响，导致我们没有必要在先验信息的分布上大动干戈。但是如果数据量不大并且我们对先验信息可能的分布形态比较自信，那么我们可以设定一个较为主观的先验分布。

4.4 对比朴素贝叶斯和*k* 近邻

在刚才的模型中，α和β两个参数通常称作“超参数”或者“伪计数”。数据科学家对选择什么样的超参数值具有完全的自主权。它们看起来似乎花哨，但只要从后验分布的角度来分析其实也十分简单。相比较而言，*k*近邻只有一个参数需要调整：近邻的个数*k*。前面的讨论中，我们发现朴素贝叶斯的终极模型形态跟线性回归模型十分相似。没错，它确实是一个线性模型（在这里是一个线性分类器），但是*k*近邻却不是。*k*近邻模型还会受到“高维诅咒”的挑战，而朴素贝叶斯却没有这个苦恼。*k*近邻的方法不需要加以训练，拿来即用；而朴素贝叶斯模型却需要多加以训练以持续改善它的模型效果。然而，它们的共同之处在于：它们都属于监督性学习模型：在建模之前我们已经知道哪些是垃圾邮件而哪些是正常邮件。也就是说，我们事先已经知道了真理，而真理监督着我们把模型做好。

4.5 Bash代码示例

```
#!/bin/bash
#
# 文件名：enron_naive_bayes.sh
#
# 文件描述：训练一个简单只包含一个单词的朴素贝叶斯垃圾邮件过滤器
# 使用的是安然（Enron）公司提供的数据集
#
# usage: ./enron_naive_bayes.sh <word>
```

```
#
# 要求：
# wget
#
# 作者：jake hofman (gmail: jhofman)
#
# 如何使用这段代码
if [ $# -eq 1 ]
    then
    word=$1
else
    echo "usage: enron_naive_bayes.sh <word>"
    exit
fi

# 如果文件不存在，则去网上下载得到
if ! [ -e enron1.tar.gz ]
    then
    wget 'http://www.aueb.gr/users/ion/data/
    enron-spam/preprocessed/enron1.tar.gz'
fi

# 如果文件夹本身不存在，解压缩刚才得到的.tar.gz文件
if ! [ -d enron1 ]
    then
    tar zxvf enron1.tar.gz
fi

# 切换到enron1目录
cd enron1

# 计算得到总垃圾邮件个数，总正常邮件个数以及总邮件个数
Nspam=`ls -l spam/*.txt | wc -l`
Nham=`ls -l ham/*.txt | wc -l`
Ntot=$Nspam+$Nham

echo $Nspam spam examples
echo $Nham ham examples

# 计算得到垃圾邮件和正常邮件中包含该单词的邮件个数
Nword_spam=`grep -il $word spam/*.txt | wc -l`
Nword_ham=`grep -il $word ham/*.txt | wc -l`

echo $Nword_spam "spam examples containing $word"
echo $Nword_ham "ham examples containing $word"

# 使用bash里的bc计算器计算相关概率值
Pspam=`echo "scale=4; $Nspam / ($Nspam+$Nham)" | bc`
Pham=`echo "scale=4; 1-$Pspam" | bc`
echo
echo "estimated P(spam) =" $Pspam
echo "estimated P(ham) =" $Pham
```

```
Pword_spam=`echo "scale=4; $Nword_spam / $Nspam" | bc`
Pword_ham=`echo "scale=4; $Nword_ham / $Nham" | bc`
echo "estimated P($word|spam) =" $Pword_spam
echo "estimated P($word|ham) =" $Pword_ham

Pspam_word=`echo "scale=4; $Pword_spam*$Pspam" | bc`
Pham_word=`echo "scale=4; $Pword_ham*$Pham" | bc`
Pword=`echo "scale=4; $Pspam_word+$Pham_word" | bc`
Pspam_word=`echo "scale=4; $Pspam_word / $Pword" | bc`
echo
echo "P(spam|$word) =" $Pspam_word

# 返回到之前的目录
cd ..
```

4.6　网页抓取：API和其他工具

作为数据科学家，数据并不总是现成的。你经常需要自己去想搜集数据的方法，如提出问题，并尝试研究和解决它。API就是采集数据的工具之一。API全称是“应用程序编程接口”，网站会通过API为开发者提供网站使用的数据。这些数据通常都具有固定的格式以便于后期处理。（API的用处还有很多，数据接口只是其中一小部分。）一般来说，使用API之前需要注册并获取一个“密钥”。密钥其实就是一串长长的密码，用来识别你已经注册并申请使用API。很多大型网站都会提供API，其中比较著名的有《纽约时报》的官方网站（http://developer.nytimes.com/docs）。



关于使用API的警告：在使用之前要仔细阅读网站关于API使用的条款细则。另外，一些网站对API的使用设定了诸多限制，包括你能获取的数据类型以及免费账户的访问频率等。

API数据的格式多种多样，业界还没有统一的API数据格式标准，因此从不同的网站得到的数据格式可能不尽相同。其中JSON是较为常见的一种。

幸运的是，我们可以使用雅虎的YQL语言（http://developer.yahoo.com/yql/）整合不同的API数据格式。你所要做的就是进入雅虎开发者网络，用类似下方的SQL语法抓取一些常见网站

的API数据，YQL会将它们整合输出成统一的格式。Python可以识别YQL的输出格式并一次性读取所有的数据。

```
select * from flickr.photos.search where text="Cat"
and api_key="lksdjflskjdfsldkfj" limit 10
```

这是标准输出，你只需要在Python中解析一次。

如果你想抓取的网页没有提供API那该怎么办呢？

在Firefox火狐浏览器中有一款名叫Firebug的插件，它能够完整的扫描网页并提取网页HTML中你想要的信息。技术上来看，Firebug就是把网页转换成了可供浏览和编辑的HTML文本。也就是说，HTML类似于一张完整的网页地图，而Firebug就是你的导游。

当你在HTML文本内找到想要的内容之后，再用诸如curl、wget、grep、awk、perl等编程工具写一两行简短的代码，想要的东西就到手了。我们其实还可以用Python或者R把上述操作全部自动化。

另外还有一些不错的文本解析工具，此处我们提及几个。

- lynx以及lynx--dump（http://lynx.browser.org/）：如果你比较怀旧，这一款应该非常适合你。它的界面还保持着20世纪70年代的风格。额……也许没有那么久远，但说是1992年的风格应该不为过。
- Beautiful Soup（http://www.crummy.com/software/BeautifulSoup/）：非常稳健但速度很慢。
- Mechanize（http://mechanize.rubyforge.org/Mechanize.html或https://pypi.python.org/pypi/mechanize/）：此款非常酷炫，但是不支持JavaScript。
- PostScript（https://en.wikipedia.org/wiki/PostScript）：适用于图像解析与分类。

> 思维实验：图像识别
>
> 如何训练计算机识别一张照片的内容是风景照还是人物肖像照呢？
>
> 首要的问题是，怎么才能弄到建模用的贴好标签的数据呢？如已经有了这些照片，可以雇一些员工，人工审核这些照片并给它们贴上标签。这似乎不太现实。其实一个简单的方式是从flickr网站（http://

www.flickr.com/ ）上抓取一些已经被用户贴好标签的照片。

取得数据之后，接下来要对照片进行数据预处理。每张照片都可以表示为一个RGB（红绿蓝）密度直方图。也就是照片中的每一个像素点的色彩都可以表示为该点的三原色组合的密度值。红绿蓝是三个基本色（也成为三基色或者三原色），其中每个颜色的强度大小都可以用一个介于0和255之间的数值表示。而所有的颜色都可以表示为红绿蓝三基色的某个特定的强度组合，也就是一个RGB密度直方图。

一个图像可以有三个密度直方图，分别为R、G和B；对应于红、绿、蓝三基色在该图像上的密度分布。因为255是个较大的数值，传统的直方图表示会变得较为杂乱。因此我们可以用块状直方图替代，这样我们需要表示的密度范围就缩减为0~51了。也就是说，一张图像可以表示为15个数值，因为我们总共需要表示3种基本颜色，而每种颜色又被细分成了5个色块。当然，我们在这里假设了每张照片的像素点个数是相同的。

最后，我们可以使用k近邻模型进行分类。比如说，我们可以挑选“蓝色”作为分类条件，k近邻很可能会根据蓝色分布上的不同将风景照和人物肖像照成功分离开来。

4.7 Jake的练习题：文章分类问题中的朴素贝叶斯模型

该题的主要目的是把朴素贝叶斯模型应用到一个多标签文本的分类问题。首先，利用《纽约时报》官网的API从《纽约时报》网站的不同栏目中抓取一些近期的文章，再把每一篇文章转换成单词向量。我们的模型任务是：给定一篇文章的单词向量，预测这篇文章可能来自于《纽约时报》的哪一个栏目？现在让我们一步一步完成这项任务。

第一步，你需要去《纽约时报》官网注册，拿到“key”之后就可以获得下载文章的API权限了。事先仔细阅读一下API的使用须知，了解基本的API权限，再编写程序分别从“艺术”“商业”“讣告”“体育”和“国际”等5个栏目中下载2000篇最近一段时间发表的文章。（提示：下载

文章时一定要注明文章到底来自哪个栏目板块，可以用nytd_section_facet这样的标签值以示区别。）你可以使用它们提供的控制台快速地了解一下该API的特性。编写代码的时候还需要注意，来自于不同栏目的文章需用单独的文件夹存储。格式方面，可以选择制表符定界格式，并在第一列上存储文章的URL，第二列上存储本章的标题，第三列上存储文章的正文部分。

下载和整理完文章的数据之后，请编写程序实现一个最基本的朴素贝叶斯模型。所有的文章都可能出自C个栏目中某一个，用y_i表示第i篇文章的栏目标签，那么$y_i \in 0, 1, 2, \ldots, C$。比如，第0类代表“艺术”，第1类代表“商业”……由此类推。每篇文章用一个稀疏的二元矩阵X表示，$X_{ij} = 1$则代表第*i*篇文章包含第*j*个单词。

先要通过计数估算出两个关键参数的取值：

$$\hat{\theta}_{jc} = \frac{n_{jc} + \alpha - 1}{n_c + \alpha + \beta - 2}$$
$$\hat{\theta}_c = \frac{n_c}{n}$$

其中n_{jc}代表*c*栏目中出现了单词j的文章数；n_c代表c栏目中的总文章数，*n*代表所有的文章数。α和β前文已经有过介绍，它们是拉普拉斯平滑估计中的两个“超参数”。有了这些参数的估计值之后，就可以给文章*x*分类了。之前的讨论中，我们已经推导出了朴素贝叶斯的最终模型形态，它与线性回归非常相似。如果我们把栏目0看作基类，那么很容易计算出其他栏目类别相对此基类的对数发生比（log-odds）：

$$\log\left(\frac{p(y=c|x)}{p(y=0|x)}\right) = \sum_j \hat{w}_{jc} x_j + \hat{w}_{0c}$$

其中：

$$\hat{w}_{jc} = \log \frac{\hat{\theta}_{jc}(1-\hat{\theta}_{j0})}{\hat{\theta}_{j0}(1-\hat{\theta}_{jc})}$$
$$\hat{w}_{j0} = \sum_j \log \frac{1-\hat{\theta}_{jc}}{1-\hat{\theta}_{j0}} + \log \frac{\hat{\theta}_c}{\hat{\theta}_0}$$

对于每篇文章的分类，模型代码工作的流程大致是这样的：首先读取

每篇文章的标题和正文，移除一些无关的字符（譬如标点符号），文本内容分词化并过滤掉一些停用词。该流程的主要目的是将文本中对于分类有用的词语解析并存储起来。在训练模型阶段，这些解析过的词语特征变量，联合我们主观设定的α和β值，便可以计算出分类模型中最关键的权重参数$\hat{\omega}$。模型训练好之后，一篇待分类文章（当然每篇文章都需要经过上述的读取、移除、分词和过滤流程，以得到一样的词语特征变量）的词语特征变量会交给已经训练好的模型，模型会输出对应每个栏目类别的后验概率。

通常我们会将手中的数据随机分成两半，一半用来训练模型一半用来检测模型的预测效果：包括预测准确度和模型运行速度。因为α和β的设定较为直观，因此需要根据模型在测试集上的预测效果不断调试。对于一篇文章，模型会给出每个类别的后验概率值。所以我们会把具有最大概率值的类别分配给该文章。模型的分类效果可以用一个5×5的混淆矩阵表示。单词的分类能力以及文章的可分类型同样也非常有意思，比如说我们可以根据模型的预测效果列出“10个分类能力最强的单词”，或者“10篇最难分类的文章”等。

如果把刚刚的模型应用到其他数据集上效果会如何呢？比如说，应用到《纽约时报》早些时候的文章，或者是其他报纸的文章数据等？这称作模型的扩展性，你可以思考一下。

使用《纽约时报》的API： R代码示例

```
# 作者：Jared Lander
#
# 用硬编码写的API请求
theCall <- "http://api.nytimes.com/svc/search/v1/
article?format=json&query=nytd_section_facet:
[Sports]&fields=url,title,body&rank=newest&offset=0
&api-key=Your_Key_Here"

# 此处我们需要rjson、plyr和RTextTools软件包
require(plyr)
require(rjson)
require(RTextTools)

## 首先让我们看一个单独的API请求
res1 <- fromJSON(file=theCall)
# 结果多长？
length(res1$results)
#查看第一项
res1$results[[1]]
# 第一项的标题
res1$results[[1]]$title
# 得到的第一个结果被转换成了数据框（data.frame），并且可以在R的数据查看器中方便地展示
```

```
View(as.data.frame(res1$results[[1]]))

# 把请求的结果转换成数据框的形式，该数据框应该是10行3列
resList1 <- ldply(res1$results, as.data.frame)
View(resList1)

## 接下来就可以应用到更多的请求了
# 创建一个字符串用来替换第一个%s和补偿第二个%s
theCall <- "http://api.nytimes.com/svc/search/v1/ article?format=json&quer
# 生成一个空的列表（list）对象以存储3个请求的记过
resultsSports <- vector("list", 3)
## 在0到2的三个数值中循环迭代得到每一个AIP请求的值
for (i in 0:2)
{
    # 创建一个查询字符串，把第一个%s替换为Sports
    # 把第二个%s替换为当前的i值
    tempCall <- sprintf(theCall, "Sports", i)
    # 应用该查询操作并得到相应的json返回值
    tempJson <- fromJSON(file=tempCall)
    # 把得到的json对象转换成一个10x3的数据框
    # 并保存为一个列表对象
    resultsSports[[i + 1]] <- ldply(tempJson$results,
    as.data.frame)
}
# 将该列表对象转换成数据框
resultsDFSports <- ldply(resultsSports)
# 创建一个新列，以表示该结果来自于Sports栏目
resultsDFSports$Section <- "Sports"

## 上述操作都是跟对Sports栏目的，也同样可以应用到arts（艺术）栏目
## 此处的代码只做参考
resultsArts <- vector("list", 3)
for (i in 0:2)
{
    tempCall <- sprintf(theCall, "Arts", i)
    tempJson <- fromJSON(file=tempCall)
    resultsArts[[i + 1]] <- ldply(tempJson$results,
    as.data.frame)
}
resultsDFArts <- ldply(resultsArts)
resultsDFArts$Section <- "Arts"

# 将上述两个栏目的结果整合到一个数据框中
resultBig <- rbind(resultsDFArts, resultsDFSports)
dim(resultBig)
View(resultBig)

## 现在进行“标记化”（tokenizing）操作
# 创建一个英文的文献-检索词矩阵（document-term matrix），剔除其中的数字和停用词，提
doc_matrix <- create_matrix(resultBig$body, language="english",
doc_matrix
View(as.matrix(doc_matrix))
```

```
# 此处分别创建训练和测试数据集
theOrder <- sample(60)
container <- create_container(matrix=doc_matrix, labels=resultBig$Section,
```



自然语言处理之历史背景

最近十分热门的自然语言处理（NLP）是计算机科学的一大研究前沿。刚刚讨论过的文本识别问题只是该领域研究中的冰山一角。NLP可以解决的问题很多，包括机器翻译、语义分析、词性标注以及文本识别等。其中的机器翻译，通过算法将一种语言自动且实时的翻译成另一种语言。而本章讨论的垃圾邮件过滤问题恰好是一个文本识别问题。NLP的研究甚至可以追溯到20世纪50年代。

第 5 章　逻辑回归

本章的贡献者是Brian Dalessandro。Brian是Media6Degree（M6D）公司分管数据科学部门的副总裁，在学术界他也十分活跃。他还兼任KDD数据科学竞赛的联合主席。M6D是一家从事在线广告业务的创业公司，其总部位于纽约。图5-1是Brian的数据科学知识构成，*y*轴上的值被卡通化成了小丑（对应小值）和摇滚明星（对应大值）。

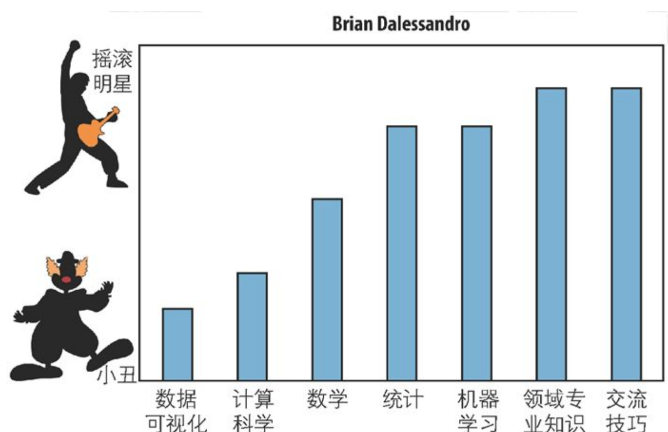


图5-1：Brian的数据科学知识构成图

Brian曾来到我们的课堂为大家上了一课， 主要内容是逻辑回归和评价。但在正式开始之前，他先做了两个思维实验。

5.1　思维实验

1. 如果“大统一理论”真的成立，那么数据科学到底还有什么特别之处呢？假设“大统一理论”指的是对世界上万事万物运行规律的普适解释。这个问题引申出了一系列问题。

- 如果真有一个普适的理论，那么我们还需要研究数据科学这样的具体学科吗？
- “大统一理论”有没有存在的可能性？这个理论如果存在，那么它是否只存在于某一领域，比如说物理学？物理学是关于世界如何运转的学科，它强调精确性，比如说可以精确预见100年才出现一次的那颗彗星何时重返。

- 如果这个理论不可能存在，就说明物理学和数据科学是有本质区别的，那这种区别是什么？

- 两者的区别就只有准确度这一项吗？或者更广义地说，我们所能想到的东西，到底有多少能分别用这两种理论来解释？是不是因为我们在预测人类行为时，研究对象的行为会受到预测本身的影响，从而形成了一种反馈回路？

  若将科学看作一个统一的整体，可能对解答上述疑问会有所帮助。在这个统一体中，精确的物理学处于最右端，而越往左走就越混乱——研究者要面对更多的不确定性和随机性（也意味着更高的薪水）。那么诸如经济学、营销学和金融学这些学科又在科学体系中处于什么位置呢？

  如果数据科学像物理学一样，已经有一套业已成熟的建模方式，那么要知道人们在何时会点击什么样的广告，就变得和预测火星探测器何时着陆一样容易。鉴于此，人们目前形成了普遍共识：无论是现在还是未来，我们都无法彻底了解这个世界。

2. 数据科学值得称作“科学”吗？

不要低估了创意的力量——很多时候人们有了设想，却未能找到实现手段。而作为数据科学家，你应该有能力把设想转化为一个数学模型，这个模型在操作上会具备一些约束条件。你需要明确地知道问题所在，快速测度问题的方方面面，并且对它进行优化。而至关重要的一点是，在建模完成之后你要确保这个模型能够解决最初提出的问题。

数据科学中也是讲究艺术的，这主要体现在将人类实际问题和数学语言互为翻译转化的过程中。

经验告诉我们，这种转化问题的方式是没有标准答案的——可选的模型总是不止一种，相应的模型评价指标也有很多，甚至连最优化的方法都有很多选择。而数据科学之所以称作科学——给定原始数据、限制条件和问题描述——其恰恰在于这样的问题总是没有绝对普适的答案，我们需要经历一个迷宫一样的过程才能找到一个可能的最优解。每一种方案的选择都可以被视作一种假设，你需要具备利用精确的测试和实验方法来检验（验真或者证伪）这些假设的能力。

这样一种假设和检验的循环往复的过程给“数据科学”深深地烙上了“科

学”的印记。具体来说，其“科学”的一面主要体现在下面三点。

- 如果你找到了一个最优的模型，坚持使用它！
- 如果你有一个新主意，把它与你之前的最优模型进行比较。通常，你需要思考一下如何设计好一对比较实验。
- 在能够100%确定之前，不停地实验（但也要尽量避免过拟合）。

5.2　分类器

本节的重点是如何选择一个好的分类器。分类器也就是分类模型：给定一些数据，它可以输出数据对应的某个类别或者是某些类别的概率值。上一章我们讲到的朴素贝叶斯模型以及k近邻都属于分类器的一种。表5-1中列出了一些需要分类器的常见情形。

表**5-1**：分类器的例子：想要回答的问题以及答案的形态

问题	答案
[illegible]	0或者1
这个图像是哪个数字	0, 1, 2[illegible]
这篇新闻来自哪个栏目	[illegible]
这是一封垃圾邮件吗	0或者1
这种药物治疗头疼有效吗	0或者1

让我们先讨论以下最简单的分类器：二元分类器（输出值为0或者1）。

本章讨论的模型叫作逻辑回归，其他的二元分类模型还包括决策树（第7章）、随机森林（第7章）、支持向量机以及神经网络等（本书没有涉及）。问题的背景是，给定一些数据和一个来自真实世界的分类问题，你需要决定：

1. 使用哪一个分类器；

2. 应用何种优化方法训练分类器；

3. 选择什么样的损失函数；

4. 哪些特征变量对建模有用；

5. 如何评估模型的实际效果。

我们先讨论第一个问题：既然有这么多的分类器可以使用，那么我们到底应该选择哪一个呢？一个自然的想法是，每一个都尝试一下，然后从所有的备选模型中选出一个最好的。这样的笨办法是不可取的，因为在现实问题中总是有各种各样的限制条件，比如数据量有限，或者时间有限等。我们没有条件做大海捞针的工作。模型选择的问题往往不受重视，人们以为总是可以在众多模型中找到一个适用的模型，其实不然。下面我们就讨论一下现实问题中一些常见的限制条件。

5.2.1 运行时间

在M6D公司，我们每天要更新500个决策模型，因此我们格外重视模型的运行和更新速度。这里的速度，指的不仅仅是更新一个模型的速度，还包括模型真实的应用速度。后者我们通常称为模型的“运行时间”，它通常比前者要更加重要。

某些模型需要大量的“运行时间”，比如说k近邻模型：当模型数据空间很大时，预测一个新数据的类别需要计算这个数据点的k个“邻居”，因此需要把所有的新旧数据点都存储在内存中，这通常会耗费大量的“运行时间”。

而线性模型则不然，无论是模型更新还是用作实际预测，它的速度通常都令人满意。下一章你将看到，线性模型的更新过程只涉及新的数据，因此不需要把旧的数据也放在内存中，这极大地提高了模型的运行速度。一旦线性模型的参数估计完毕，只需要保存这些参数的估计值，预测新数据只涉及计算参数估计向量与新数据特征变量的点积的问题。

5.2.2 你自己

有一个限制条件常常被我们忽视，那就是我们自己！在应用模型分析数据的时候，要不时地拷问自己，是不是真的理解你正在使用的模型。

在这一点上，我们要诚实一点。如果不理解也没有关系，因为没有人能理解所有的模型，这也并不是成为一个数据科学家的必要条件。但是通常来说，如果你想发挥一个模型的最大效用，必须全面理解这个模型，了解模型参数的含义、模型的假设条件等方方面面。有时，你需要调整模型算法来拟合自己的数据。由于未能透彻理解手中的模型，人们经常不知不觉就陷入了过拟合的陷阱。

5.2.3 模型的可解释性

商用模型需要具备良好的可解释性，不然客户不会买账。有些模型天生可以说故事，比如决策树模型；而有些模型则是个黑匣子，你根本不知道里面发生了什么，比如随机森林模型。即便随机森林只是决策树模型的一个推广，在可解释性方面却有天壤之别。复杂的模型通常都很难解释，因此如果你没有足够的时间去解释一个复杂模型的结果，可以考虑牺牲一些准确度，把模型稍作简化。

一个典型的例子就是信用评分模型。法律规定，信用卡公司如果拒绝一个信用卡申请，必须给出充分的理由。 因此在信用评分模型中，决策树模型要比随机森林模型更加适用。法律没有规定所有的模型都要具有很好的解释性，但如果能把模型很好地解释给同事或者客户，何乐而不为呢？

5.2.4 可扩展性

制约模型可扩展性的是模型的成本，现实中会考虑以下三方面的成本。

1. 学习时间：也就是模型的训练时间。

2. 得分时间：也就是模型的预测时间，给定一个数据和已经训练好的模型，新用户要多久才能得到一个预测值？

3. 模型存储：模型运行时要占用多大的内存？

“An Emprical Comparison of Supervised Learning Algorithms”（“监督学习算法的比较研究：基于实证的观点”，http://goo.gl/bLpoea ）是一篇关于算法比较的很好的文章，读一读你会学到以下几点。

1. 模型的复杂程度经常与它的精度成正比。简单的模型可能具有更好的可解释性，但可能不会有令人满意的预测精度。

2. 没有万能算法，每个问题都要根据其自身的特点选择最适合的模型。

3. 诸多条件会限制着你对可用算法的选择，包括数据量大小，项目成本和时间成本等。

5.3 逻辑回归：一个来自M6D的真实案例研

究

在M6D，Brian和他的团队亟须解决以下三个核心问题。

1. 特征工程：找出最有用的特征变量并且知道如何正确处理和使用它们。

2. 交互预测：模型需要具备迅速的用户响应能力，用户鼠标按下的一瞬间，模型要即时地预测和输出。

3. 精准定价：若给定广告能够向目标用户展示，可以产生多大价值？

5.3.1　点击模型

对于M6D公司来说，工作的核心就是根据客户的要求找到广告的最佳受众。也就是说，他们需要计算在电脑前点击鼠标的你，有多大的可能点击他们所要展示的广告。那么对于这样一种商业模式，需要分析什么样的数据呢？他们应该如何使用模型来提高广告的点击度呢？

M6D会追踪客户访问的每一个网页，但是数据科学家并不会挨个分析网页的具体内容，而只需要知道网页对应的URL地址。他们会将这些地址表示为一些随机的字符串，随着用户的网页访问历史不断积累，这些字符串会累积成一个字符串向量，代表用户访问的完整历史。譬如说用户u在一段时间内访问了四个网页，那么可以用下面的字符串表示他的URL访问历史：

```
u = <&ltfxyz, 123, sdqwe, 13ms&gtg>
```

其中每个子字符串都代表其访问过的某个网页的URL。如果每个用户的访问历史都用这样的字符串向量表示，那么全部用户的访问历史可以合并成一个巨大的矩阵：矩阵的行代表用户，列代表其访问过的网站。如果矩阵的某个元素值为1，则代表相应的用户访问过相应的网页。可以想象，这样的一个矩阵将会十分稀疏，也就是说，会有大量的0值存在。这是因为网站访问是一个非常个性化的行为，每个人访问过的网页都不尽相同。

如果这个数据是被用作一个分类的问题，我们至少需要一个分类变量作为被解释变量。对于点击模型来说，可以假设有一条卖鞋的广告，而我们想要分析人们是否点击了此条广告。此处的分类变量就是该条广告是否被用户点击过，它是一个二元变量值：1代表被某用户点击过，0则表示没有被该用户点击。当我们设定好一个被解释变量之

后，加上之前已经处理好的稀疏特征矩阵，数据的准备工作就完成了。换句话说，一个训练数据集已经准备完毕。

接下来的任务就要交给科学家了：搭建模型、在数据集上训练模型等。第3章的垃圾邮件过滤例子中，训练数据集中的特征变量是单词。具体说来，那里是代表单词是否出现在某封邮件中的一个二元变量，因此当时的模型并不关心单词的真实含义。现在，你可以依赖已为垃圾邮件检测构建好的算法。

此处的情形其实十分相似，我们并不关心某个网页的真实内容，而只关心用户是否访问了该网页。因此朴素贝叶斯模型也同样适用。然而，还有另外一个模型也同样适用于该情形，那就是逻辑回归模型，它是本章的绝对主角！

一言以蔽之，逻辑回归的输出值是用户点击某广告的概率值。这和朴素贝叶斯模型的输出值如出一辙。因此，在得到模型的概率输出值之后，由你决定输出的值到底应该是1还是0。此处的惯用手法是设定一个阈值（比如说0.75），只要输出的概率值大于此阈值则输出1，否则输出0。这种输出方法与传统的线性回归模型有着本质的不同：传统的线性回归模型会输出任何一个介于负无穷和正无穷之间的实数值，而逻辑回归的输出值是实实在在的概率值，它位于0和1之间。因此可以说逻辑回归是为了分类模型而生的模型。

前一章已经提醒过大家，如果你生搬硬套地使用传统线性回归模型，比如说你把数据交给R，R会义无反顾地输出模型的结果而不管你使用的模型是否合适。线性回归模型的输出值并不是一个概率值，你可能会得到一个小于0的值，或者是一个大于1的值。

5.3.2　模型背后

逻辑回归的微妙之处就在于输出值是介于0和1之间的概率值，那么到底为什么这样呢？在这里，我们把模型背后的数学稍作呈现，希望可以加深读者对逻辑回归的认识。唯一的奥妙就在于，数据的特征矩阵被某一个神奇的函数巧妙地转换成了一个严格位于[0, 1]之间的数值，那么这到底是一个什么样的函数呢？从微积分的角度来看，我们需要找一个函数，其定义域为全体实数，而值域为[0, 1]。反逻辑函数（inverse-logit function）就是这样一个函数。下式是该函数的表达形式，图5-2是该函数的图像：

$$p(t) = \text{logit}^{-1}(p) = \frac{1}{1+e^{-t}} = \frac{e^t}{1+e^t}$$

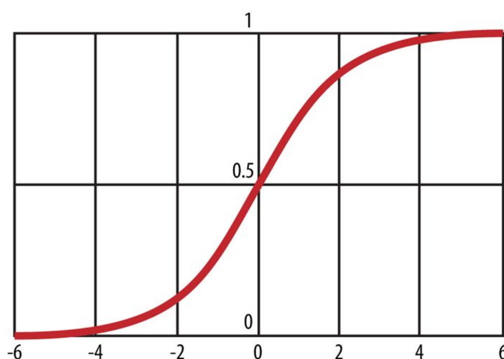


图5-2:反逻辑函数图

逻辑函数与反逻辑函数

逻辑函数的定义域是[0, 1]，而其值域为整个实数集。用p表示函数的变量，则逻辑函数可以表示为：

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right) = \log(p) - \log(1-p)$$

顾名思义，反逻辑函数就是逻辑函数的反函数，因此其定义域为整个实数集合，而值域为[0, 1]。

从函数的表达式来看，t值越大e^{-t}值越小，因此分母越接近于1，$p(t)$也越接近于1。同样，t值越小，则e^{-t}越大，因此分母的值也越大，$p(t)$越接近于0。这样的一个反逻辑函数是逻辑回归的灵魂。当然，真正的逻辑回归表达式也比函数本身要复杂一点：

$$p(c_i|x_i) = [\text{logit}^{-1}(\alpha + \beta^T x_i)]^{c_i} \cdot [1 - \text{logit}^{-1}(\alpha + \beta^T x_i)]^{1-c_i}$$

式中，c_i代表分类变量（被解释变量）的标签值（点击为1，没有点击则为0），x_i是用户i的特征变量。注意，c_i是一个二元变量值，当其取1时，上式可简化为：

$$p(c_i = 1|x_i) = \frac{1}{1+e^{-(\alpha+\beta^T x_i)}} = \text{logit}^{-1}\left(\alpha + \beta^T x_i\right)$$

同理，当其取值为0时：

$$p(c_i = 0|x_i) = 1 - \text{logit}^{-1}\left(\alpha + \beta^T x_i\right)$$

对上面两种简式作商，可以得到类似线性模型的表达式：

$$\frac{p(c_i = 1|x_i)}{p(c_i = 0|x_i)} = \alpha + \beta^T x_i$$

也可以用刚才提到的逻辑函数来表示，那么模型的最终形式为：

$$\text{logit}\left(p(c_i = 1|x_i)\right) = \alpha + \beta^T x_i$$

如果你觉得我们是在绕圈圈，那么你的感觉是对的！绕圈圈的目的是告诉你，逻辑回归的最终形式可以巧妙地转化成线性模型的形式，而模型的输出结果却被限制在了0和1之间。

有了上面的最终表达形式，我们便可以给逻辑回归一个明确的定义了：对于点击模型来说，用户i点击卖鞋广告的逻辑概率可以用该用户网页访问历史的特征变量（也就是之前对URL预处理得到的稀疏矩阵）的线性组合来表示。

模型中的α称作基准值，也就是在对用户的访问行为一无所知的情况下对用户点击广告概率的无条件猜测。从均值的含义来理解，它代表了用户群体点击广告的平均概率值。因为我们其实对此知之甚少，所以它的值一般都很小，通常近似1%。从概率论的角度来看，它代表无条件概率值。

如果你不了解自己的特定情况，只知道基准值，那么预测值的均值只与α相关：

$$p(c_i = 1) = \frac{1}{1 + e^{-\alpha}}$$

而β是模型的斜率值，它一般是一个向量，其长度和模型的特征变量个数相同。β向量中的每一个值都代表了相应特征变量对模型输出概率值的相对贡献程度。

5.3.3 α和β的参数估计

在应用模型预测之前，首先要估计模型中的参数α和β。传统线性回归

中的参数估计方式并不适用于逻辑回归，因为逻辑回归的似然函数的最大化问题没有解析解，不能通过传统的求导方式解决。但是，由于其似然函数是一个典型的凸函数，因此可以用凸优化的方法找到最优解。

用Θ表示α和β的参数组合，L代表模型的似然函数，则有：

$$L\Theta|X_1, X_2, \cdots, X_n) = P(X|\Theta) = P(X_1|\Theta), \cdots, P(X_n|\Theta)$$

此处我们假设数据点X_i是相互独立的，$i = 1, ..., n$代表每一位用户。这里的独立性假设表明一位用户的点击行为与另外一位用户毫不相干——在这里的例子中，用户的“点击行为”指的是用户的”点击概率“。该假设虽然并不代表真实情形，但通常来说是可以被接受的。（在上式中，我们之所以将似然函数写成概率密度函数连乘积的形式也是因为独立性假设。）

给定了数据之后，我们需要找到一组α和β最大化上面的似然函数。观察数据可知：

$$\Theta_{MLE} = argmax_{\Theta} \prod_{1}^{n} P(X_i|\Theta)$$

令$p_i = 1/\left(1 + e^{-(\alpha+\beta^T x_i)}\right)$，其代表某一个观测值的概率值，那么一个数据点的概率密度$P(X_i|\Theta)$为：

$$p_i^{c_i} \cdot (1 - p_i)_i^{1-c_i}$$

假设独立性假设成立，那么参数的最大似然估计可以写作：

$$\Theta_{MLE} = argmax_{\Theta} \prod_{1}^{n} p_i^{c_i} \cdot (1 - p_i)_i^{1-c_i}$$

那么现在的问题是，如何最大化似然函数呢？

对于线性回归来说，可以应用微积分中的极值定理找到能够最大化似然函数的参数组合：分别对α和β取一阶导数并令其为0，然后检查相应的二阶偏导数是否小于0。然而，对于逻辑回归来说这招却是行不通的。这下该如何是好呢？解决办法是有的：先把似然函数写成对数似然函数，因为对数函数是一个单调递增函数，因此最大化似然函数

等同于最大化对数似然函数。然后在对数似然函数前取负号，得到负对数似然函数。这样我们就把一个最大化问题变成了一个最小化问题。取负号是因为负对数似然函数是一个凸函数，因此可以用凸优化的方法找到最小化负对数似然函数的最优解，也就是我们想要的参数组合α和β。

凸优化是一个较为成熟的研究领域，有很多现成的算法可以使用，我们需要决定到底使用哪个算法。接下来我们将介绍其中两种较为常见的。通常情况下，这两种方法都可以收敛到全局最优解。这里的“通常情况”指的是数据的变量之间相互独立，从而保证优化过程中的海塞矩阵（Hessian matrix）满足其正定性条件。



关于最大似然估计

刚才的似然函数部分我们讲得可能太快了。如果你对最大似然估计很感兴趣，我们建议你翻一翻Casella和Berge合著的*Statistical Inference*（《统计推断》）一书。如果你对其中涉及线性代数的部分感到吃力，我们建议你把Gilbert Strange教授的著作*Linear Algebra and Its Applications*（《线性代数及其应用》）仔细研读一番。

5.3.4　牛顿法

微积分中的牛顿法是函数优化的经典方法，我们可以尝试用它找到对数似然函数的全局最优解。在微积分中，我们知道一个函数的泰勒展开式的前几项可以很好地近似该函数本身，牛顿法就是基于此原理。

具体来说，我们定义$\nabla\Theta$为函数的局部梯度值（也就是在某点上的一阶导数），H表示海塞矩阵（也就是函数的二阶导数矩阵），那么牛顿法会一步一步（迭代）地找到最优解，其迭代的步幅用γ表示，迭代公式为：

$$\Theta_{n+1} = \Theta_n - \gamma H^{-1} \cdot \nabla\Theta$$

可以看出，牛顿法参数最优化的过程中，其每一步迭代的方向都与对数似然函数的曲度保持一致，也就是说每一步迭代的目的都是向极值点不断地靠近。由于式中涉及海塞矩阵（$k * k$）的求逆操作，因此牛

顿法在数据量较大、变量较多的情况下（如10 000）会遇到计算瓶颈。但是大多数情况下也不会出现变量过多的情况。

其实在实际计算中，我们很少直接对海塞矩阵求逆，而是通过解一个类似$Ax = y$的线性方程组间接地找到A的逆（A^{-1}），这通常比直接求逆矩阵要容易得多。

5.3.5　随机梯度下降法

随机梯度下降法（http://en.wikipedia.org/wiki/Stochastic_gradient_descent）是求似然函数极值的另一个算法。前面已经提过梯度，指的是函数在某一点处的一阶导数的值，也就是函数的变化率。随机梯度下降是一个顺序算法，每次只关注一个数据点，每迭代至下一个数据点都会根据所得到的信息最优地更新参数的估计值，以此类推直到穷尽所有的数据值。这相比于之前的牛顿法，优点在于它不需要对矩阵求逆。通常就这一点就可以完爆其他的算法，因为它可以有效地适用于大数据和稀疏特征矩阵的情形。因此也被类似于Mahout（http://mahout.apache.org/）和Vowpal Wabbit（http://hunch.net/~vw/）这样的开源机器学习软件包所重点收纳。当然，它也有缺点：它的优化效果有时候不会太好，并且十分依赖于步幅的设定。

5.3.6　操练

逻辑回归的参数估计比传统回归更适用于分类模型，但是其参数的估计方法，包括迭代加权最小二乘法和随机梯度下降等方法，都要比普通最小二乘方法复杂得多。然而，你并不需要自己编程实现这些参数的具体估计过程，类似R这样的统计软件已经把这些方法都收入麾下了，你所要做的只是找到适合的函数并应用到手头的数据上即可。比如说，我们手上有一个小数据集，只有5行观测值和5个URL标签。

```
click url_1 url_2 url_3 url_4 url_5
1       0     0     0     1     0
1       0     1     1     0     1
0       1     0     0     1     0
1       0     0     0     0     0
1       1     0     1     0     1
```

我们把这个数据矩阵用`train`命名，那么在R中建立一个逻辑回归模型基本上可以用下面的一行代码解决：

```
fit <- glm(click ~ url_1 + url_2 + url_3 + url_4 + url_5,
```

```
            data = train, family = binomial(logit))
```

5.3.7 模型评价

本章一开始曾经指出，要得到一个效果良好的分类模型，数据科学家需要考虑诸多因素。其中，选择模型评价标准就是一个十分重要的因素。第3章和第4章讨论过了如何评价线性回归模型、*k*近邻以及朴素贝叶斯模型。但是总体来说，模型的评价没有一个统一标准，它既取决于分析的数据，也和模型本身有关。因此在选择的时候需要十分谨慎。就拿逻辑回归来说，它既可以用于对二元变量的建模，也同样适用于多标签变量，因此在选择模型评价方法时，需根据不同问题采取不同方法。

首先，若将逻辑回归用在排序模型中，比如说你想将广告或商品被用户点击的概率进行排序。你可以先利用逻辑回归估算各个概率，把模型的输出结果从大到小排序。如果建模者对输出类别的相对排序感兴趣（而不是个别类的绝对概率值），那么适用的模型评价方法包括下面两个。

- 操作者特征曲线面积（**ROC**面积）

  前面我们提到，逻辑回归的输出值是一个概率值。如果我们想得到一个二元输出值，那么可以定义一个阈值。对于一个分类模型的分类效果，通常我们会比较关注两个指标：真阳性值（实际类别为1，预测类别也是1）和伪阳性值（预测类别是1，而实际类别为0）。操作者特征曲线（也就是ROC曲线）结合了阈值和两个阳性值指标，最早是被信号检测学家用来选择最佳信号的检测模型。该曲线图的纵轴代表模型的真阳性值，横轴代表模型的伪阳性值。给定一个逻辑回归模型和一个阈值，模型分类效果的真阳性值和伪阳性值都可以方便地计算出来，并可以绘制成一个二元平面上的点。阈值可以在-∞和∞之间连续变动，因此代表（真阳性值，伪阳性值）的点也可以连续变动，既而形成一条曲线。这条曲线就称作操作者特征曲线，它描绘了模型的分类效果与阈值的变动关系。该曲线到横轴之间的面积通常称作操作者特征曲线面积（AUC），它是评价一个分类模型效果的经典指标，因此也可以用来比较两个分类模型孰优孰劣。想知道更多关于ROC和AUC的细节，你可以参考Tom Fawcett的文章“Introduction to ROC Analysis”（“ROC分析导论”，参见http://goo.gl/fE8i7s）。

- 累积提升图

在直销行业，建模者较多地使用累积提升图判断一个模型到底有没有用。模型有用的最低标准是，它的效果至少应该好于随机猜测。

在第13章中，我们还将用到这两种图形评价方法。

模型产品化的问题

如果你想把逻辑回归产品化并应用于广告排序的实际问题上，那你应该仔细审视所用数据的生成过程。这可能说得有点抽象，让我们用一个实例来说明。譬如说，在给用户展示广告时，你将某发胶的广告放在了香体露广告的上方，然后发现发胶广告获得了更多的用户点击率，那么这究竟意味着发胶广告更受用户欢迎呢，还是因为你把它放在了更加显眼的位置呢？也就是说，广告的排列位置可能会潜在地影响广告本身的点击率，而广告本身的质量就变得难以考证了。[1] 解决“干扰因子”影响的办法就是把该因子本身作为预测变量加入到模型当中。在建模过程中，干扰因子的影响十分复杂却又至关重要，我们也很难三言两语就说明白。可以想象，在谷歌甚至存在一“广告质量控制部”，专门负责研究和解决类似的问题。

1在统计学中，这通常称作“干扰因子”。

其次，假设要将逻辑回归模型用于分类问题，那么我们已经知道数据的实际输出是一个二元值（0, 1）， 而逻辑回归的模型输出是一个介于0和1之间的概率值。为了将逻辑回归模型应用到没有标签值的样本，我们需要将逻辑回归的实际输出（一系列概率值）转换成二元值（0和1）。为了最小化模型的误分率，我们需要在这个转换过程中选择一个合适的阈值。比如说阈值为0.5，那么只要预测概率值大于0.5， 预测标签值就为1（代表被点击）；反之则为0。模型的评价有很多指标，我们在第3章和第4章也讨论了一些。我们把这些已经讨论过的模型评价标准拿出来，再结合一些没有讨论过的标准，在这里一起呈现给大家。你会想，如此多的标准，在实际应用中到底应该选择哪一个呢？其实，每一个标准都各有侧重，对它们的选择也要因地制宜。

- 提升度
  提升度指的是分类模型的预测精度相比随机猜测模型的提升幅

度。

- 准确度
  模型的准备度指的是所有被正确预测的类别（包括阳性类和阴性类）。

- 精确度
  模型的精度指的是模型的真阳性值/所有的阳性值。也就是说，在所有被预测为阳性的类中，其真实为阳性的比例。

- 召回率
  召回率指的是模型的真阳性值/（模型的真阳性值 + 模型的假阴性值）。也就是，在所有应该被预测为阳性的类中，其真实被预测为阳性的比例。

  ***F*得分**
  之前我们没有提到任何关于*F*得分的内容，它是精度和召回度的调和平均值，其形式为：(2 * 精确度 * 召回度) / (精确度 + 召回度)。可以看出，它综合考虑到了精确度和召回度的不同特性。*F*得分还有很多变种，其区别主要在于精确度和召回度的权重不同。

最后，如果我们想评价或比较模型输出的实际概率值的精确程度，可以用以下三种评价标准。

- 均方误差
  之前关于线性回归的章节中提到过均方误差的概念，它指的是实际值和预测值之间的平均平方距离。

- 根均方误差
  顾名思义，根均方误差就是均方误差的平方根。

- 平均绝对离差
  与均方误差不同的是，平均绝对离差计算的是预测值和实际值之间的绝对值距离，而不是平方距离。

综合来看，AUC（接受者操作特征曲线下面积）要比提升曲线更适合用作模型比较。提升曲线的一大特征就是："基率不变性。"比如说，如果模型将用户点击率从1%提升到2%，其提升率为100%；而如果从4%提升到7%，其提升率反而小于100%。然而，很明显后者的提升效率更高。从这一点来看，AUC要更加适用于模型的比较。

我们有必要再讨论一下输出概率模型与输出排序模型之间的区别。假设在广告点击模型中，每条广告的成本是c元，而如果该广告被用户点击（用conversion表示），可得收益为q元。若想获取利润，那么相应的收益必须大于成本，也就是说：

$$p(\text{conversion}|X) \cdot qgt; c$$

可以看到式子的最左边代表用户点击某广告的概率值大小。因此，在利润要求下，能够准确估算出广告被用户点击的概率值对于保证利润有着举足轻重的作用。在这里，广告排序（谁更有可能被用户点击）的意义就相形见绌了。比如说，某三条广告的排序为1、2、3，代表第一条广告最有可能被用户点击，其对应的概率值可能为0.7、0.5和0.3。然而，即使对应的概率值只有0.03、0.02和0.01，它们之间的排序关系也不会有任何改变。但是这么小的概率值对于利润的影响却是巨大的。因此，类似均方误差的评价标准更适合用在此处。

> 在模型评价中应用**A/B**测试优化法
>
> 当我们选择某个模型评价标准（比如前面提及的准确度、均方误差等），并据此建立和优化模型的时候，从根本上来说，我们是在找模型中相关参数的最优解。有时候对模型的评价适宜使用复合的标准，例如前面谈到的利润标准，而利润标准的本质其实还是模型的概率输出的准确度。因此模型本身很难直接反映我们想要的最优结果：最大化的利润。为了克服这个缺点，通常的做法是应用A/B测试优化法。举例来说，我们想比较两个模型哪个更优，那么我们可以随机地给两个模型组分配两组用户，并分别计算两组用户产生的利润值。这里需要注意的是，我们直接使用了我们最为关心的“利润值”指标，而并非模型准确度、均方误差这样的模型类指标，因此我们得以直接考察两组模型在创造利润上的差距，从而决定使用哪个模型。这就是统计实验设计学中的A/B测试优化法，我们将在第11章做深入探讨。

5.4　练习题

M6D慷慨地提供了一个数据集供我们练习逻辑回归的建模、分析和评价。数据可以从https://github.com/oreillymedia/doing_data_science直接下载。下面的代码可作参考。

示例R代码

```
# 作者：Brian Dalessandro
# 读取数据，检查变量，创建训练和测试数据集
file <- "binary_class_dataset.txt"
set <- read.table(file, header = TRUE, sep = "\t",
        row.names = "client_id")
names(set)

split <- .65
set["rand"] <- runif(nrow(set))
train <- set[(set$rand <= split), ]
test <- set[(set$rand > split), ]
set$Y <- set$Y_BUY

##########################################################
########### R 函数 ##########
##########################################################

library(mgcv)

# GAM平滑图函数
plotrel <- function(x, y, b, title) {
        # 对数据生成一个GAM平滑表示
        g <- gam(as.formula("y ~ x"), family = "binomial",
        data = set)
        xs <- seq(min(x), max(x), length = 200)
        p <- predict(g, newdata = data.frame(x = xs),
        type = "response")
        # 现在得到实证分析的结果（如果结果不是离散化的，还要进行离散化操作）
        if (length(unique(x)) > b) {
            div <- floor(max(x) / b)
            x_b <- floor(x / div) * div
            c <- table(x_b, y)
        }
        else { c <- table(x, y) }
        pact <- c[ , 2]/(c[ , 1]+c[, 2])
        cnt <- c[ , 1]+c[ , 2]
        xd <- as.integer(rownames(c))
        plot(xs, p, type="l", main=title,
                ylab = "P(Conversion | Ad, X)", xlab="X")
                points(xd, pact, type="p", col="red")
        rug(x+runif(length(x)))
}

library(plyr)

# wMAE图函数及其相关计算
getmae <- function(p, y, b, title, doplot) {
        # 将数据正则化到[0, 1]区间范围内
        max_p <- max(p)
        p_norm <- p / max_p
```

```
        #细分成b个间隔
        bin <- max_p * floor(p_norm * b) / b
        d <- data.frame(bin, p, y)
        t <- table(bin)
        summ <- ddply(d, .(bin), summarise, mean_p = mean(p),
                mean_y = mean(y))
        fin <- data.frame(bin = summ$bin, mean_p = summ$mean_p,
        mean_y = summ$mean_y, t)
        # 计算wMAE
        num = 0
        den = 0
        for (i in c(1:nrow(fin))) {
            num <- num + fin$Freq[i] * abs(fin$mean_p[i] -
                fin$mean_y[i])
            den <- den + fin$Freq[i]
        }
        wmae <- num / den
        if (doplot == 1) {
            plot(summ$bin, summ$mean_p, type = "p",
                main = paste(title," MAE =", wmae),
                col = "blue", ylab = "P(C | AD, X)",
                xlab = "P(C | AD, X)")
        points(summ$bin, summ$mean_y, type = "p", col = "red")
        rug(p)
    }
return(wmae)
}

library(ROCR)
get_auc <- function(ind, y) {
        pred <- prediction(ind, y)
        perf <- performance(pred, 'auc', fpr.stop = 1)
        auc <- as.numeric(substr(slot(perf, "y.values"), 1, 8),
            double)
    return(auc)
}

# 针对某个特征变量集合，用x交叉验证法评估模型表现

getxval <- function(vars, data, folds, mae_bins) {
        # 将观测值分配到相应的交叉验证分组中
        data["fold"] <- floor(runif(nrow(data)) * folds) + 1
        auc <- c()
        wmae <- c()
        fold <- c()
        # 生成一个公式（formula）对象
        f = as.formula(paste("Y", "~", paste(vars,
            collapse = "+")))
        for (i in c(1:folds)) {
            train <- data[(data$fold != i), ]
            test <- data[(data$fold == i), ]
            mod_x <- glm(f, data=train, family = binomial(logit))
            p <- predict(mod_x, newdata = test, type = "response")
```

```
            # 计算wMAE
            wmae <- c(wmae, getmae(p, test$Y, mae_bins,
                "dummy", 0))
            fold <- c(fold, i)
            auc <- c(auc, get_auc(p, test$Y))
        }
    return(data.frame(fold, wmae, auc))
}

################################################################
########## 主程序：模型与作图 ##########
################################################################
# 现在在模型中加入所有可用变量
# 检查模型的估计参数和拟合效果

vlist  <-  c("AT_BUY_BOOLEAN",  "AT_FREQ_BUY",
        "AT_FREQ_LAST24_BUY",
        "AT_FREQ_LAST24_SV", "AT_FREQ_SV", "EXPECTED_TIME_BUY",
        "EXPECTED_TIME_SV", "LAST_BUY", "LAST_SV", "num_checkins")
f = as.formula(paste("Y_BUY", "~" , paste(vlist,
        collapse = "+")))
fit <- glm(f, data = train, family = binomial(logit))
summary(fit)

# 计算每个变量对应的模型评估值
vlist  <-  c("AT_BUY_BOOLEAN",  "AT_FREQ_BUY",
        "AT_FREQ_LAST24_BUY",
        "AT_FREQ_LAST24_SV", "AT_FREQ_SV", "EXPECTED_TIME_BUY",
        "EXPECTED_TIME_SV", "LAST_BUY", "LAST_SV", "num_checkins")

# 生成一个空向量用来存储模型表现评估结果
auc_mu <- c()
auc_sig <- c()
mae_mu <- c()
mae_sig <- c()

for (i in c(1:length(vlist))) {
    a <- getxval(c(vlist[i]), set, 10, 100)
    auc_mu <- c(auc_mu, mean(a$auc))
    auc_sig <- c(auc_sig, sd(a$auc))
    mae_mu <- c(mae_mu, mean(a$wmae))
    mae_sig <- c(mae_sig, sd(a$wmae))
}

univar <- data.frame(vlist, auc_mu, auc_sig, mae_mu, mae_sig)

# 画出每个变量的MAE图

use holdout group for evaluation
set <- read.table(file, header = TRUE, sep = "\t",
            row.names="client_id")
names(set)
```

```
split<-.65
set["rand"] <- runif(nrow(set))
train <- set[(set$rand <= split), ]
test <- set[(set$rand > split), ]
set$Y <- set$Y_BUY

fit <- glm(Y_BUY ~ num_checkins, data = train,
            family = binomial(logit))
y <- test$Y_BUY
p <- predict(fit, newdata = test, type = "response")

getmae(p,y,50,"num_checkins",1)

# 向前贪婪选择法
rvars <- c("LAST_SV", "AT_FREQ_SV", "AT_FREQ_BUY",
        "AT_BUY_BOOLEAN", "LAST_BUY", "AT_FREQ_LAST24_SV",
        "EXPECTED_TIME_SV", "num_checkins",
        "EXPECTED_TIME_BUY", "AT_FREQ_LAST24_BUY")
# 生成一些空向量
auc_mu <- c()
auc_sig <- c()
mae_mu <- c()
mae_sig <- c()

for (i in c(1:length(rvars))) {
        vars <- rvars[1:i]
        vars
        a <- getxval(vars, set, 10, 100)
        auc_mu <- c(auc_mu, mean(a$auc))
        auc_sig <- c(auc_sig, sd(a$auc))
        mae_mu <- c(mae_mu, mean(a$wmae))
        mae_sig <- c(mae_sig, sd(a$wmae))
}
kvar<-data.frame(auc_mu, auc_sig, mae_mu, mae_sig)

# 画出三个AUC曲线
y <- test$Y_BUY
fit <- glm(Y_BUY~LAST_SV, data=train,
        family = binomial(logit))
        p1 <- predict(fit, newdata=test, type="response")
fit <- glm(Y_BUY~LAST_BUY, data=train,
        family = binomial(logit))
p2 <- predict(fit, newdata=test, type="response")
fit <- glm(Y_BUY~num_checkins, data=train,
        family = binomial(logit))
p3 <- predict(fit, newdata=test,type="response")

pred <- prediction(p1,y)
perf1 <- performance(pred,'tpr','fpr')
pred <- prediction(p2,y)
perf2 <- performance(pred,'tpr','fpr')
pred <- prediction(p3,y)
perf3 <- performance(pred,'tpr','fpr')
```

```
plot(perf1, color="blue", main="LAST_SV (blue),
        LAST_BUY (red), num_checkins (green)")
plot(perf2, col="red", add=TRUE)
plot(perf3, col="green", add=TRUE)
```

第 6 章　时间戳数据与金融建模

本章的贡献者为来自GetGlue公司的Kyle Teague以及我们的老熟人Cathy O'Neill。Cathy所讲的内容涉及时间序列分析、金融建模以及fancypants regression。但在学习这此主题之前，Kyle将与我们一起分享他关于推荐系统方面的经验。（关于这一主题，另见第7章。）最后，我们会了解一下关于时间戳数据的基本知识，这与Cathy所要讲的内容前后呼应。

6.1　Kyle Teague与GetGlue公司

Kyle Teague是GetGlue公司分管数据科学与工程的副总裁。Kyle早先从事于电子工程领域。他认为自己之所以成为一个数据科学家，与他当年在研究室和实验室里所做的大量关于信号处理方面的工作不无关系。他从很小的时候就开始接触编程，现在他主要使用Python语言。

GetGlue同样是一家总部位于纽约的初创公司，它们的主营业务是关于电影电视的内容开发。传统上，我们搜寻自己感兴趣节目的方法是参考报纸/电视台手册上的频道列表。这样的传统最早可以追溯到19世纪50年代。而现在，电视上有成千上万的频道以及难以计数的电视和电影节目，用传统的方法找起来似乎有点不太现实了。这就是电影电视内容开发/推荐所要解决的问题。

GetGlue想要改变传统搜寻电视节目的方式，提供给人们定制化的节目推荐方案。具体来说，每当用户收看某个电视节目，他们相应地制造了一条时间戳数据，也就是说他们在某个时点进行了“收看”这一动作。当然，“收看”并不是唯一的动作，用户可以“赞”某个节目，或者对某节目加以“评论”等。

用户的这种时间戳行为会被存储在一个类似于｛用户, 动作, 节目｝的三元向量中。图6-1称作对偶图，可以用来描绘这种类型的数据：

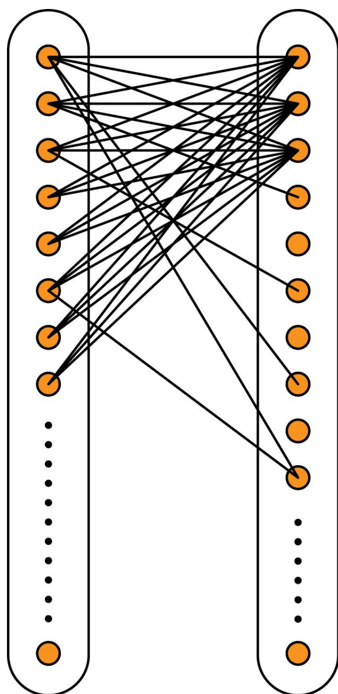


图6-1：对偶图，左图的实心点代表用户（user），右图中的实心点代表节目（item）

图中的实心点称作“节点”，左图的节点代表用户，右图的节点代表节目；节点之间的线称作“连接线”。之所以称作“对偶图”是因为图中左右各有两个“试管”，其中的节点通过连接线相互联系。如果左图的某个用户节点与右图的某个节目节点之间有连接线，则代表该用户对该节目有过反馈。比如说，某用户点赞了某节目，则该用户和节目之间就用连接线相连。需要注意的是，“试管”内部的节点之间不存在连接线。

GetGlue公司突破了试管内部节点直接无连接的限制，并假设内部节点也是可以相连的。比如说，用户A可以加用户B为好友，那么A节点和B节点之间就形成了一个有向的连接线，通常用一个带有箭头的直线表示。即使A与B用户没有直接互粉，GetGlue公司也可以通过用户的资料特征猜测某两个用户可能具有类似的欣赏品味，进而把用户A推荐给用户B（当然也会把用户B推荐给用户A）。

刚才所讲的是用户之间的连接，而节目之间同样可以根据相似性相互连接，这项工作通常需要人工来做，而GetGlue公司就雇了不少员工

专门从事这项工作。比如说，电影*TrueBlood*（《真爱如血》）和*Buffy the Vampire Slayer*（《魔法奇兵》）从某个角度来看非常相似，那么工作人员就会在表示这两个电影的节点上画上连接线，以表示它们互相之间有相似性。但是如何确定连接线是否带有箭头，带单向箭头还是双向箭头就比较微妙了。通常的做法是按照时间先后顺序。比如说，《真爱如血》在《魔法奇兵》之前上映，那么箭头的方向就从前者指向后者。《真爱如血》当年票房一片红火，这也可能意味着《魔法奇兵》也会同样受欢迎，因为它们都是关于吸血鬼的电影。因此，这里的箭头不仅仅代表了电影内容的相似性，也可能意味着他们受欢迎程度的相似性。著名的在线电台公司Pandora据说也在产品中使用类似的标注法。

在这个行业，推荐的时效性是最为重要！很多的电视节目，过了这个村可能就没有这个寨了；而用户的“点赞”行为也是一闪而过，因此对数据的搜集要讲求一个字：“快”，并且每个数据点的时间戳也至关重要，这就是意味着我们要把数据的形式增加到四元：{用户，动作，节目，时间戳}。记录时间戳信息可以帮助我们掌握用户喜好的传播特征，从而更好地因地、因时的为用户推荐他们最想看的电视节目。

6.2 时间戳

带有时间戳的数据是大数据时代的典型特征之一，它促进了大数据时代的到来。因为人们在使用计算过程中的每个动作其实都会被计算机精准地记录下来，尤其是这些动作发生的时点。这就意味着，即使是单个用户在一天之内也可以产生巨大的数据量。比如说，你在网站上的每次点击，或者每次使用软件，甚至是打电话，都会被计算机（或者手机/平板等移动设备）记录下来：包括这些动作发生的时点、持续的时间和动作的具体内容等。在软件业，新产品或者产品的新功能发布后，软件工程师一般都会在软件内部放置一段记录用户行为的代码，这段代码也是产品的一个重要组成部分，用来侦测用户对软件的使用情况以期不断改善该产品的用户体验。

比如说，在用户访问《纽约时报》的主页时，网站会记录用户被呈现了哪些文章，而用户实际点击并阅读了哪些文章等， 这样跟对一个客户，会有大量的日志文件（也就是时间戳数据）产生。

下面是一段来自GetGlue的公司的用户原始数据：


```
{"userId": "rachelschutt", "numCheckins": "1",
"modelName": "movies", "title": "Collaborator",
"source": "http://getglue.com/stickers/tribeca_film/
```

```
collaborator_coming_soon", "numReplies": "0",
"app": "GetGlue", "lastCheckin": "true",
"timestamp": "2012-05-18T14:15:40Z",
"director": "martin donovan", "verb": "watching",
"key": "rachelschutt/2012-05-18T14:15:40Z",
"others": "97", "displayName": "Rachel Schutt",
"lastModified": "2012-05-18T14:15:43Z",
"objectKey": "movies/collaborator/martin_donovan",
"action": "watching"}
```

我们可以按照之前讨论的四元数据结构把相应的字段提取出来，`{"rachelschutt", "action": "watching", "title":"Collaborator", timestamp:"2012-05-18T14:15:40Z" }`就代表{用户， 动作，节目， 时间戳}，其中用户为rachelschutt，动作是“观看”，节目是一个题为“合作者”（Collaborator）的文章，时间发生在2012年5月18日下午2点15分40秒。

6.2.1　探索性数据分析（EDA）

第2章我们已经重点提出，在建模之前最好充分地利用探索性数据分析的方法彻底了解手中的数据。这里针对用户的行为数据，我们再深入讨论一下探索性数据分析的方法。EDA是一项富有灵活性的工作，不同的数据类型、不同的研究目的，相应都需要有不同的探索思路，因此没有统一的、纲领性的EDA工具，数据科学家要学会灵活应对。

对于用户行为数据分析来说，EDA要做的第一件事就是画出每个用户的使用行为时间序列图。想掌握用户的整体特征，前提是做足够细致的局部分析工作。研究者要从单个用户的角度去挖掘数据所阐释的意义，从而确保所收集的数据是合情合理的，但这并不意味着你需要审查完每一个用户。

先在所有用户中取出一个样本，通常不需要太多用户，100个即可。也许产品有上百万的用户，但为了先搞清楚状况，你得先挨个研究单一客户的使用行为。仔细审查上百万用户的数据是不可能完成的任务，同样也没有必要。通常看完100个用户的数据之后，你对用户整体特征应该已经了解透彻。但是，如果想更精细地研究和推断用户行为特征，100个数据肯定不足，而且你还可能需要模型的帮助。

假设你随机选取了100个用户，并把他们各自用一个时间序列图表示了出来，图6-2展示了4位用户的时间序列图。

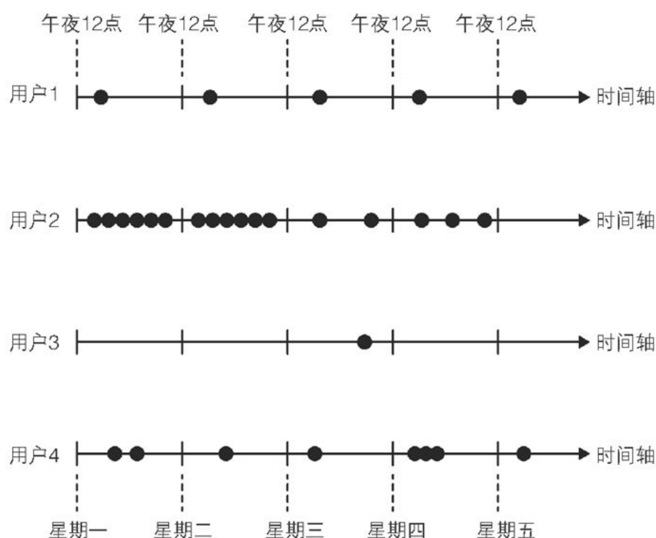


图6-2：随机选取的四位用户的时间序列图

从这个图中，我们可以发现什么呢？很明显，用户1每天使用登录的时点都基本固定，而用户2在刚开始频繁地登录了数次，之后的登录次数迅速减少，用户3在整个时间段只登录了一次，因此我们可能观察更久的时间才能了解该用户的登录特征，而用户4每天的登录时间和次数都不太固定，这可能是一个“正常用户”，这里的“正常“与否其实相当主观，你也可以说用户2是正常的。

看完了100个用户的时间序列图之后，可以问自己下面几个问题。

- 用户的典型或者平均特征是怎样的？
- 用户之间的行为有哪些明显的差异？
- 根据用户的特征，有可能将他们分门别类吗？
- 怎样将用户之间的行为差异定量化以便后续研究？

要回答这些问题，首先我们要和原始数据打交道。原始数据与图中呈现的样子大相径庭，所以你首先要解决的问题是如何把原数据组织成有效的形态，以便可以用类似图6-2的时间序列图表示和研究它们。从图中可以看出，每个用户的时间戳的个数，出现的时间都是不同的，这样的数据清理起来会比较麻烦。

比如说用户的行动有4种可能，即“点赞”“点衰”“喜欢”以及“评论”，那么这样的用户行为怎么用类似图6-2的时间序列图绘制出来呢？从数据本身的角度来说，我们应该如何存储这些数据？图6-3提供了一种可能，相比于图6-2，该图使用了两种颜色表示用户的“点赞”和“点衰”动作，其中红色表示前者，黑色表示后者。

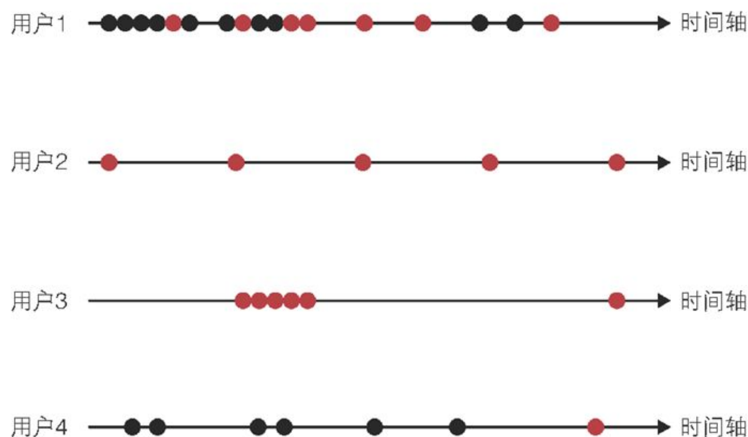


图6-3：在用户的时间序列图中，用不同的颜色代表用户不同的动作类型。红色表示“点赞”，黑色表示“点衰”

上图中一个有趣的现象是在时间序列图的最后一个时点，所有用户都点了赞。如果发现一个类似这样的明显特征，我们最好停下来想一想到底有没有蹊跷的地方。比如说，这到底是数据本身的特征，还是系统出现了故障。如果可能是后者，我们如何确定它属于系统故障。像这种用户群体同时发生的动作在其他地方还有没有出现过？黑色的节点是否普遍要多于红色节点？用户是偏向于喜欢“点赞”还是“点衰”？是否某些用户群倾向于“点赞”而一些用户群则倾向于“点衰”，而另外一些用户群则更倾向于属于“混合型”用户？怎么定义用户的“混合型”行为？可见，能引起我们兴趣的问题很多。

完成单一用户的核查之后，下一步就要想一想怎么对用户进行汇总归类。比如说，图6-4就是一个汇总的时间序列图，图的最下方横轴表示时间，纵轴表示所有用户的总计数。

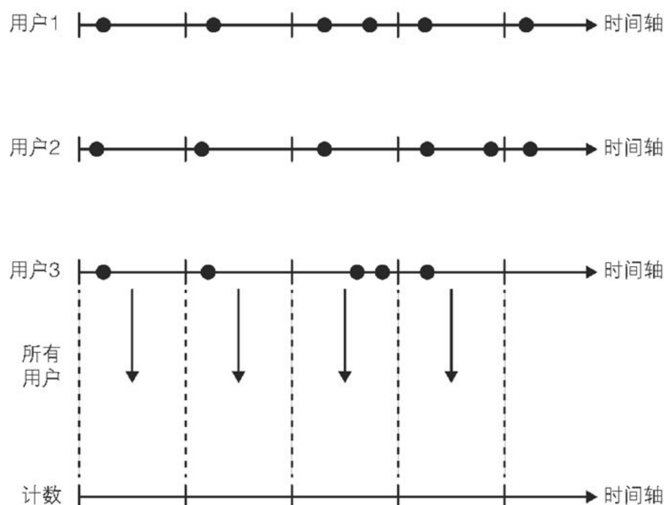


图6-4：用户汇总时间序列图

现在面对的是全体用户的数据，我们选择了一个汇总统计量以研究总体用户的平均行为特征。对于时间戳数据来说，即便是汇总统计量的选取也不会那么容易。因为用户可以随心所愿地登录，因此我们到底是汇总用户数还是用户的登录次数呢？又比如说，在某个特定的时间段内，是汇总用户的总行为数还是汇总进行了某项行为的总用户数？这样的抉择问题在时间戳数据处理和分析中十分常见。

现在让我们再回到时间序列图的本身，问自己一个更为基本的问题：这个图横轴的时间到底代表什么？是以秒计算，还是以天计算？如果选择了以天计算，那原因为何？图中是否有明显的趋势或者波动特征，波动特征是否明显掩盖了趋势特征？类似的问题还有：我们的用户是否来自同一个时区？比如说，用户1来自纽约而用户2来自伦敦，那么纽约的早晨7点则意味着伦敦的中饭时间。如果我们的基准时区是纽约，并总结说：从图中看来，有30 000个用户在早晨7点对某电影点了赞；如果有很多其他时区的用户，那么这样的总结就会相当不准确，甚至于歪曲了事实。对于这种情形我们又该如何处理呢？此时可以根据所有用户的时区将他们的时间戳用一个基本时区（比如说，纽约时间）来表示。类似这样的问题在数据处理中还有很多，我们需要注意分辨、分析，并根据问题的实际特征加以更正。



时间戳数据数据处理的噩梦

我们也不想骗你，时间戳数据可能是你见过的最难搞定的数据类型，时区就是一个大问题。通常的做法是把所有时间点都转换成一个基准时间（参见http://www.epochconverter.com/）后再做分析。

我们可以做的事情还有很多，比如不同的行为类别可以用不同的图示方法表示出来，某些行为类别还可以合在一起构成新的分类。这些进一步的分析往往可以给数据科学家带来有关数据的不同方面和不同角度的有价值信息。

6.2.2　指标和新变量

探索完数据之后，我们可能已经对数据有了一定程度的认识，接下来的任务就是构建指标/变量了。比如说，用户点赞的次数、用户第一次点赞的时间等都可以作为模型的变量。即使是构建一些简单的二元变量也可能十分有用，比如用户“至少点赞了一次”等。构建指标和新变量的主要目的是方便我们对用户进行比较，换句话说，就是计算用户之间的相似度和相异度。在构建新指标和变量的时候，大致有两个方向可以走：一个是以用户为基准，汇总行为数；一种是以行为为基准，计算涉及的用户数。

因此，我们可以构建无数个指标或者新变量，上限就是你的想象力。但是，在构建新变量的时候，一个最基本的原则是：这些指标和变量的含义要清晰，要对理解数据有所帮助。

6.2.3　下一步怎么做

在完成探索性数据分析，并构造了一系列的新指标和变量之后，下一步当然就是建模，构思和实现算法，深入地分析数据了！但具体要怎么做还是要根据问题而定，我们这里呈现几个例子：

我们可以建立时间序列模型，并用作预测。比如说较为常见的自回归模型就是时间序列模型的一种，在下一节的金融建模中我们会详细讲解。在时间序列分析中，如果某个建模对象对时间较为敏感（比如，市场总是时涨时跌，与时间的关系密切。时间在市场分析中就扮演着

极为重要的角色），这样我们就可以根据建模对象的时间特征预测它下一步会如何表现。

我们也可以做聚类分析（第3章已经详细讨论了聚类分析的内容）。对于用户数据的聚类分析，其关键点也同样在于如何定义用户之间的相似性。

或者我们会想要构建一个用户行为预警系统：比如说某个用户的行为特征发生了异常就自动给管理人员发送警报。当然，这里我们首先需要定义何为“异常“行为？

我们也可以做一个事件监测系统，也叫作“变点分析”。也就是说我们可以根据用户的行为特征判断在某个时点发生了“不寻常事件“。更深入的问题还包括：用户的何种行为会触发类似的事件？用户的行为与该事件的发生有无因果关系等？不可否认的是，因果关系的研究难度会比较大，它需要涉及类似上一章的A/B实验的内容。还有一个终极目标，我们可能会想做一个推荐系统，给用户推荐他们想要的电视节目。虽然对于我们个人来说不太现实，但这正是GetGlue在做的事。

历史背景

时间戳数据其实并不新颖，时间序列分析本身也不是一个新兴学科。对时间序列分析感兴趣的读者可以阅读James D. Hamilton的*Time Series Analysis*（《时间序列分析》）一书。在时间序列研究伊始，数据量一般都比较小，数据发生的频率也不高：通常是日度或者月度数据。比较高频的数据最早出现，是在金融学研究中的股票数据、信用卡的交易数据、电话的通讯记录数据、图书馆的图书借阅数据等。

时间戳数据的形态正在发生着日新月异的变化。这主要体现在：首先，得益于移动设备的迅猛发展，对人们日常行为的记录变得难以置信的简单和直接。其次，时间戳具有相当的精确性，因为这些数据是由设备本身记录的，而不是由人自己汇报的。众所周知，机器不会撒谎，但个人汇报的数据却并不可靠。最后，随着计算能力的迅猛发展，机器可以储存大规模的数据，处理时间也相应较快。

6.3　轮到Cathy O'Neill了

Cathy是我们的老熟人了，她的数据科学知识构成可见图6-5。

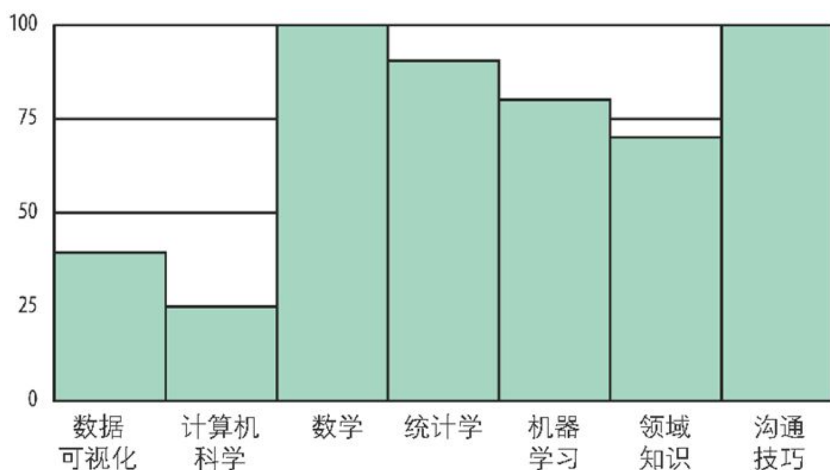


图6-5：Cathy的数据科学知识构成图

CS（计算机科学）是Cathy的明显弱项，不过她的Python编程技巧十分娴熟，可以轻松抓取和解析想要分析的数据，建立模型并使用matplotlib绘制好看的图形。然而对于使用Java进行map-reducer操作，Cathy就不太擅长了。她也同样不太擅长数据可视化，不过她很善于演讲以及与人交流。

6.4　思维实验

如果你选择忽视数据中内含的时间戳信息，你会失去什么？

答案是，你将很难梳理出数据中内含的因果关系，因为时间往往在因果关系中扮演着至关重要的角色。

但是如果我们稍微改变一下刚才的问题，假设你并没有具体的绝对的时间戳信息，但是还是收集了一下“相对时间戳”信息，比如说“离上次用户最后登录的时间”，或者“离上次用户点击的时间”等。

答案是：即使有“相对时间戳”信息也于事无补。诸如数据中的趋势和季节性特征等都会被“相对时间”忽视。以之前的胰岛素数据为例，你可能会发现在注射胰岛素15分钟后，血糖会持续降低，这是“相对时间戳”会告诉你的信息。但是如果只有这个“相对时间”而没有具体的时间戳信息，你可能会忽视这样一个长期趋势：在过去的几个月中，你

的血糖一直在持续走高。同样对于图6-6，因为每个时点的绝对时间戳都被详细地记录了下来，因此可以看到有明显的季节波动趋势。如果只有“相对时间戳”数据，你很容易忽略了季节性特征。对于金融数据来说，如果想通过观测序列的走势找到合适的买入/卖出点，记录具体的时间戳信息更显得极为重要。

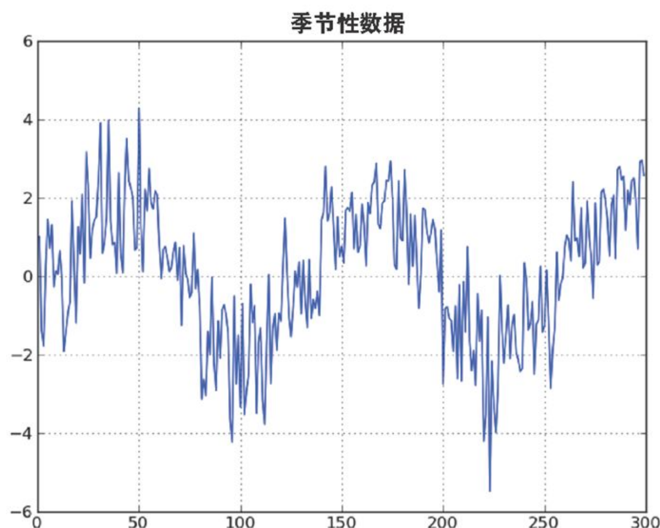


图6-6 如果不记录具体的时间戳信息，像本图中的季节性特征是无从发现的

6.5　金融建模

数据科学家的头衔出现之前，在金融领域“金融分析师”这一职业已经存在了许久。从工作性质上来看，金融分析师与数据科学家既相似又大有不同。其中一点就是，金融分析师十分在意时间戳数据，甚至有点迷恋， 他们甚至不太关心为什么数据会是这个模样，而只关心数据发生的时间点。

金融建模是一个内涵丰盈的研究领域，我们接下来就感受一下它的魅力所在。

6.5.1　样本期内外以及因果关系

首先需要定义何为“样本期内”，何为“样本期外”，对于金融建模这是一个基本的概念区分。样本期内，顾名思义就是在观测期范围内的样本数据，而样本期外与我们前面章节所讨论过的“测试数据”有所关

联，但又有本质的不用。测试数据本质上还是在观测期内，只是将观测器划分成了训练数据和测试数据两个部分，一部分用来训练模型，另一部分用来验证和测试模式。而金融建模中的样本期外的数据严格发生在建模之后，对于样本期内的数据来说，它们是未观测到的数据。其目的是用来检验金融模型在未观测数据上的真实预测表现。

此外，样本外分析的次数甚至都应该加以严格限制。因为，每一次分析，我们都将手头的样本内数据学习了一遍，如果样本外分析的次数过多，即使是研究的问题稍有不同，抑或是不同的模型，我们在潜意识里都有可能不知不觉地得到过拟合的结果。

其次，当执行因果关系建模的时候（这与统计学家所说的因果关系有所不同），我们必须多加小心。核心原则可以一语概括：永远不要拿未来的数据去预测未来！听起来似乎十分拗口。换句话说，意思也就是，预测样本期外的数据时只能使用样本期内的数据，因为样本期外对于样本期内来说是尚未观测到的。其实，更加严格的来说，即使是现时的数据，如果你还没有搜集到，也不能用在对未来的预测上。一语以蔽之：你只能使用样本期内已经观测到的数据！稍后我们会讲到一个涉及政府部门的时间序列数据。

同样，因为时间戳的存在，在给定一个训练数据集后我们难以确定何为模型的“最佳估计参数”。因为从数据的第一个时间戳开始估计，每增加一个时间戳，模型的估计都会发生相应的改变，直到估计到最后一个时间戳。也就是说，我们不会得到一个“最佳估计值”，而是得到一系列随着时间推进而演化的估计序列。

这类估计方法的有用之处在于，我们可以借此检验模型估计的稳定性。譬如说，如果某个参数的估计值的正负号在估计序列中前前后后变化了十几次，那么我们可以认为该参数的估计相当不稳定，而索性将其从模型中赶出去。当然，具体要不要把它赶出去还要看数据和模型的具体含义，也许它是一个天生就好动的参数，但在模型中却扮演着十分重要的角色。

图6-7用红色线将样本期内与样本期外分离开来，它所要表达的含义是：样本期内永远发生在样本期外之前（以避免因果关系问题）。

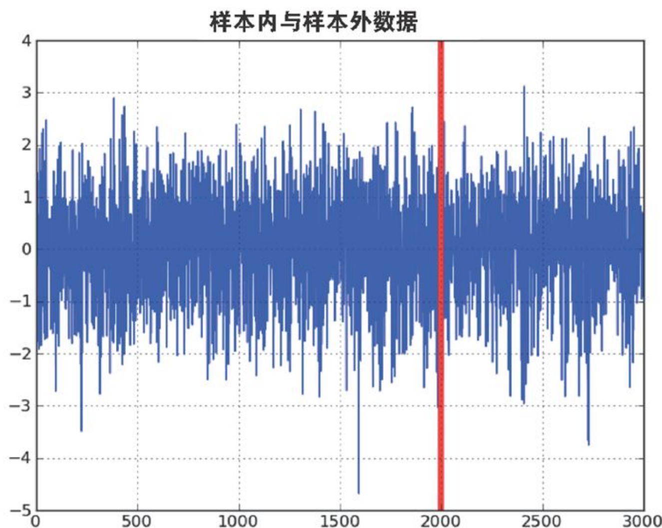


图6-7：样本期内的数据永远发生在样本期外数据之前，红色线代表了模型建立的时点

关于因果关系建模，以及样本内与样本外的关系，在建模分析中很重要，在产品代码中也应该得到一致性的体现。我们在建模的时候——无论是在训练数据上还是在做样本外模拟——都假设模型可以用在之后的实际产品当中，并发挥相应的效用。当然，模型的训练和拟合用的是样本内的数据，因此样本内的模型表现自然要好于它之后在产品中的表现。

另一种说法是，一旦建好了模型并用在产品中之后，预测和决策只能建立在预测期之前的可得数据上，这是因果建模的核心准则。然而，一旦搜集到新数据，我们可以理所当然地把新数据加入模型，更新模型的参数估计并既而更新模型的预测值。因此，随着时间的推进，模型也理应被不断地更新、优化并变得越来越好。

6.5.2　金融数据处理

在实际建模之前，通常要对数据进行预处理和分析：比如，做对数变换，计算均值和方差等。这是很好的习惯，也称作“预建模”。

数据变换

人们对于正常数据的分布的总体印象来自于正态分布，并认为数据差不多都应该是对称的钟形分布，而

许多模型恰好也假设数据服从某种形式的正态分布。然而，金融数据往往会表现“失常”，它们的分布往往既不对称，也明显不是钟形分布（比如说，分布过尖），因此需要对数据进行变换，让它们的形态回归“正常”。下面是一些常用的变换方法。

- 将数据减去其均值并除以其标准差，变换后的数值严格位于0到1之间。这称作标准化变换。
- 在标准化变换中，也可以除以数据的最大值。
- 对数变换：对变量的每个观测值取对数。
- 将数据从小到大等分成5个组，每组的观测值数量相同。
- 将数据变换成二元变量（0或者1）。最常见的变换方法我们之前有过论及，可以取一个阈值，如果变量的值大于该取值则变换为1，否则变为0。

一旦我们估计出了数据的均值$\bar{y}$和方差σ_y^2，新观测到的数据点也可以被标准化。如果原始变量服从一个正态分布，那么标准化后的变量严格服从标准正态分布：一个均值为0，标准差为1的正态分布。标准化变换可以表示为：

$$y_i \to \frac{y - \bar{y}}{\sigma_y}$$

数据预处理的方式多种多样，具体要怎么做要取决于你的具体情况和研究目的以及之后要用的一些子模型。比如说，有时候我们可能更关心数据的相对变动，也就是说，在t时点，我们想知道y_t与上一时点的观测值y_{t-1}有何区别；或者，我们希望通过一个子模型（比如一个单序列的回归模型等）找到y_{t-1}的哪部分更适合用来预测y_t，那么这个时候它们的差值$y_t - y_{t-1}$可能就是我们更加关心的变量。[1]

> 1这种差分操作在金融建模的很多场合都十分常见。

数据变化的方法同样很多，到底选择何种变换也同样取决于数据本身以及研究目的。需要铭记于心的是，时间序列的观测值是有序的，时间上的顺序代表着可能的因果关系，因此当你在训练模型以及引入新数据更新模型时一定要遵循数据的先后关系。永远不要在建模中作弊，不要使用还没有发生的样本外数据。

一个典型的例子是有关在标准化操作中均值的选用：一定不要使用整个时间段的数据均值。对于时间序列数据来说，通常要计算“移动均值”，也就是你在某个时点t上所知道的局部均值，在标准化时也应该使用该局部均值，而非全局均值。2

> 2也就是说随着时间的推移，序列的均值也在相应地发生变化。这样做的原因在于时间序列数据的变动性大，不确定性高，单一的全局均值不能体现序列的变动特征。

如果不这样做，可能会导致十分危险的结果。举个例子来说，假设数据中间段发生了市场崩溃3，这样的极端情况虽然只是偶有发生，但仍会对整个序列的均值和方差产生巨大影响。如果在均值的估计中只采用上文所说的全局均值，模型的效果从整体上来看会显得异常得好：我们似乎可以在市场崩溃之前就预测到它的到来了。这样的伪因果估计只是从形式上提升了模型的预测能力，或者让一个原本很差的模型变得看起来还不错（甚至，让一个根本没有价值的模型变得价值连城）。4

> 3金融市场的崩溃反映在时间序列上就是金融产品价格的极速下跌。

> 4在统计学中，均值是一个典型的“不稳健”统计量，它很容易受到极端值的影响，如果一个序列中存在一两个极端值，它们可以显著地影响全局均值的大小。

6.5.3　对数收益率

金融学中，收益率观测的标准频率以日计算，市场十分关注收益率的日度变化值。每天市场都会开盘和收盘，那么计算收益率的方法可以基于开盘情况也可以基于收盘情况。以开盘为例，我们可以在周一和周二的早晨记录下两天各自的开盘价格，并用周二的价格减去周一的价格即得到周二的收益率。但是，通常的做法还是以收盘价格为基准计算收益率。

相比于百分比收益率，我们用得更多的是对数收益率：假设第t日的收盘价格为F_t，那么该日的对数收益率可以表示为$\log(F_t/F_{t-1})$，相应的百分比收益率定义为$100(F_t/F_{t-1})-1$，如果去掉百分比收益率定义中的100则得到绝对百分比收益率。

相比于绝对百分比收益率，对数收益率具有可加性。也就是说，5日的对数收益率等于5天内每日对数收益率之和。可加性带来了巨大的计算便利性，也是通常人们更倾向于使用对数收益率而不是绝对百分比收益率的重要原因。

对数收益率另外一条有趣的性质是“对称性”。举例来说，一只股票的价格为2，其先降低50%则价格变为1（绝对百分比收益为-0.5），那么如果该股票的价格从1回到2，其百分比收益率为100%，绝对百分比收益为1，而-0.5和1是不对称的。如果使用对数收益率，则log(0.5) = -0.301与log(2) = 0.301是严格对称的。

然而如果数据的频率较高，比如说日度数据，或者更高频的分钟数据封，其百分比收益率与对数收益率之间的差距非常小。这个很容易证明：令$x = F_t/F_{t-1}$，则百分比收益率为$x - 1$而对数收益率为$\log(x)$。对$\log(x)$应用泰勒展开式，可以得到：

$$\log(x) = \sum_n \frac{(x-1)^n}{n} = (x-1) + (x-1)^2/2 + \cdots$$

因此对数收益率始终大于百分比收益率，但是只要x的值非常小，那么上式中右项从平方项开始会越来越小，甚至可以忽略不计。对于高频数据来说，x - 1的值通常都很小，满足忽略不计的条件。因此，对于高频数据，可以用对数收益率很好的近似百分比收益率，并且可以保留对数收益率可加性和对称性的优良特性。

图6-8的直线为百分比收益率，曲线为对数收益率，可以看出在$x = 1$附近，两条线十分接近。

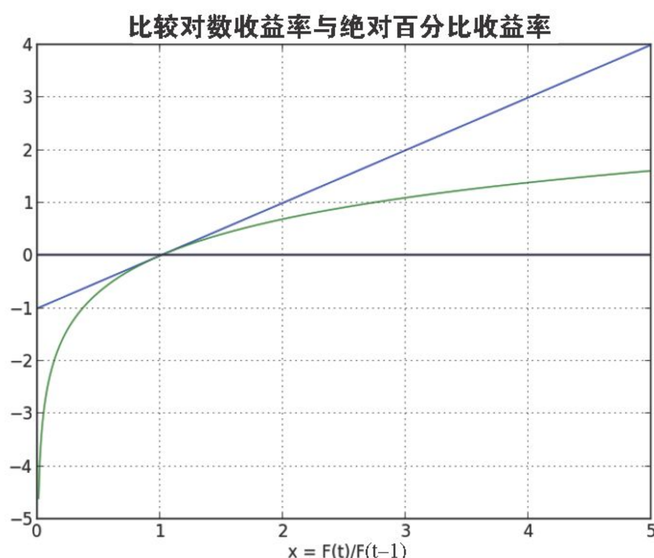


图6-8：对数和绝对百分比收益率曲线对比图

6.5.4 实例：标准普尔指数

图6-9是标准普尔指数从2001年到2012年的收盘价格指数时间序列图，图6-10为其对应的对数收益率时间序列图。

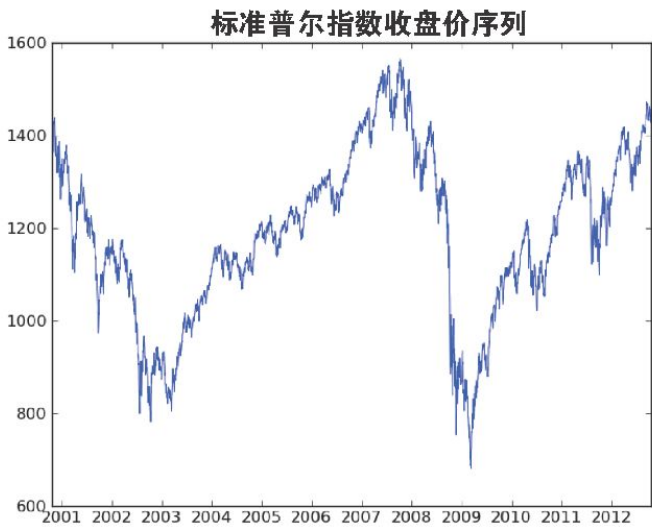


图6-9：标准普尔指数收盘价格图

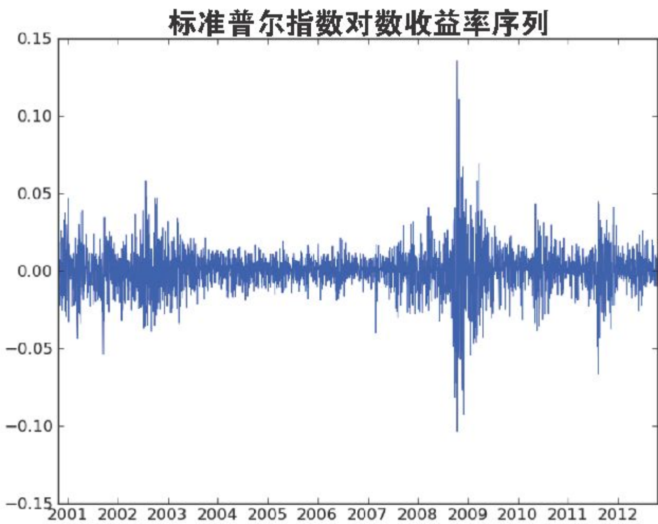


图6-10：标准普尔指数对数收益率图

图6-9看起来还似乎有规律可循，图6-10看起来则是一团乱麻。这是金融时间序列中特有的波动率特征，2008年底的金融危机带来的巨大波动率在图中表现得十分明显。如果应用之前提到的标准化数据变换操作，则可以有效地压缩对数收益率序列的波动率。图6-11就是标准化之后的对数收益率，可以看到，波动率在整个观测区间内都十分一致，没有出现图6-10中那样的巨大波动。

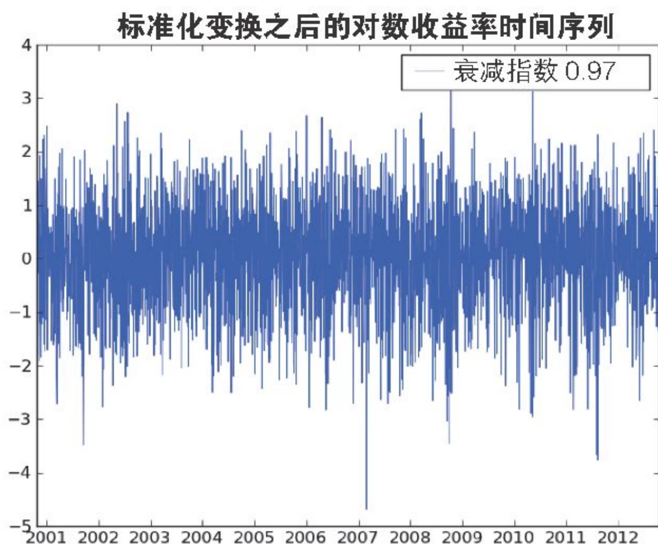


图6-11：标准化变换之后的对数收益率时间序列图

6.5.5　如何衡量波动率

定义好收益率之后，我们可以计算收益率的变化程度，也就是收益率的波动率。通常我们用收益率样本的标准差作为波动率估计。

波动率计算的关键问题在于样本回溯窗口期的设定，也就是说对于某个时点*t*的波动率，适合用多少期之前的样本来估计。回溯窗口期越长，能够用到的信息越多；窗口期越短，波动率对新样本的加入则可能更加敏感。对于窗口期，试想一下市场中发生了一个大事件，比如说金融危机，那么该事件的影响需要多长时间才能消逝？这么一段时间就类似于窗口期。这个例子虽然还是十分模糊，但窗口期大概就是那么回事。对于一个大事件来说，窗口期明显要大于一周或者一个月，但超过一个季度的窗口期则比较少见。当然，这要取决于事件本身影响力的大小。

其次，在计算收益率时，对于窗口期内的数据点，我们还要决定如何确定每个数据各自的权重大小。如果是严格地滚动窗口，也就是说窗口期内的所有数据都被赋予相同的权重，窗口期之前的数据则不会被考虑进来。但这种等权重的计算方法与通常事件的演变方式有所出入。这类似于k近邻模型，在窗口期内离计算时点越近的点往往对其影响更大，因此在计算波动率时权重的赋值可以根据时间的差距逐渐递减。

这种逐渐递减的计算方法也称作“指数平滑法”：离得越远的数据其权重会指数式得递减。这里的指数应该是一个小于1大于0的小数。比如说如果指数值为0.97，那么第五天前数据的权重就是0.97^5。这里的权重是绝对权重，在实际计算时还要除以权重值的总和以计算出每个时点上数据的相对权重。如果权重严格地按照指数递减，我们可以相应地找到窗口期内的“半衰期”对应的时点：也就是在这一点上，其对计算时点事件的影响已经下降了50%。对于0.97来说，其半衰期为-ln(2)/ln(0.97) = 23。也就是说，需要经过23个时点，计算时点事件的影响会下降一半。

确定了窗口期和平滑指数值之后，对窗口期内的每个收益率值取平方，乘以其对应的相对权重后相加取和（平方和项），再取其平方根就得到了波动率的估计值。

注意我们刚刚给大家的公式，其计算过程包含了所有之前的收益率的值，即便权重值在指数递减，这样的计算很可能还是没有止境的，这里有一个实用的小技巧：如果窗口期很大，那么只需要计算一个波动率对应的平方和项。新时点的波动率的计算，只需要将新时点对应的收益率取平方，加入之前的平方和项即可。当然，在加入平方和项之前应该乘以其对应的权重。

我们来详细地解释一下。因为所有的权重都是指数幂的形式，那么所有窗口期内收益率权重的和也称作几何平均值，如果窗口期足够大，该值为1/(1 - s)，其中s为指数值。当前时点收益率的波动率的估计值可以表示为：

$$V_{\text{old}} = (1-s)\cdot\sum_i r_i^2 s^i$$

假设在新时点的收益率为r_0，那么在该时点的波动率估计量为：

$$V_{new} = s\cdot V_{\text{old}} + (1-s)\cdot r_0^2$$

我们说过，波动率的估计量通常用窗口期数据的标准差来表示，而标准差的计算需要减去序列的均值。而我们刚才展示的计算过程似乎没有任何涉及均值的计算项。那是因为我们已经假定数据已经做了中心化操作，因此均值为0；或者我们处理的是类似日度数据等高频数据，其均值通常非常接近0。当我们计算低频数据，比如季度数据或者年度数据的波动率时，那么毫无疑问我们应该把均值的影响考虑进来。



关于指数平滑法

对于将大量数据压缩成一个估计值的问题，指数平滑法非常好用。因为，估计值的更新非常容易，我们不需要每次都重新计算一遍。



当然，用多大的指数值对于指数平滑的效果有直接影响，这里可以参考图6-12.

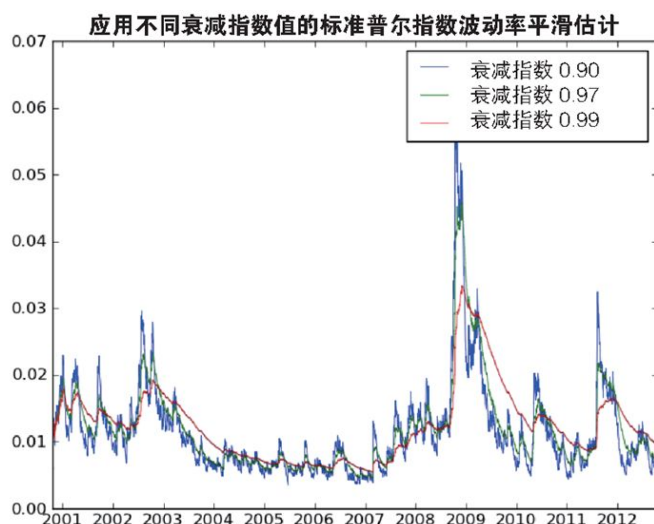


图6-12：标准普尔指数的波动率的指数平滑估计：使用了三个不同大小的指数值

其实，指数值的确定方法除了人为主观设定以外，也可以采取最小化风险函数的方法。5

> 5在工业界，通常人们都会采取后者，因为其更加客观，可以消除主观因素的影响。但是如何计算风险函数同样是个十分主观的问题。

6.5.6　指数平滑法

我们刚才用标准普尔指数的例子讨论了如何计算其收益率的波动率。这里我们更详细地介绍它的数学模型。

对于*t*时点的估计值我们假设为E_t，设定*s*为衰减参数（其通常等于1 - 指数，因此是一个很小的值）。指数平滑的基本思想是，离观测时点越近的值其权重越大，越远的值其权重越小，因此E_t的指数平滑公式为：

$$E_t = s \cdot E_{t-1} + (1 - s) \cdot e_t$$

其中e_t为在*t*时点的新观测值。



估计的可加性

我们要求每个估计值都具备可加性（对于前面提高的滚动估计来说，方差是可加的，而标准差是非可加的）。如果我们最终想要的估计值是一个加权平均值，那么加权值的分母项与分子项都需要具备可加性。

指数平滑法固然有效，但是在实际应用时还是应该小心谨慎一点儿，尤其是在数据样本量比较小的时候（比如样本量只有几百或者更少时）。与其用一个固定的平滑参数，一种更加保险的方法是使用变动的平滑参数（其实是平滑参数的倒数，但由于数与其倒数是一一对应的，所以这里不做严格区分）。可以把平滑参数的倒数与序列的半衰期的概念联系起来，刚开始半衰期为1，然后慢慢变长，直到逼近其真实的半衰期值$N = 1/S$。因此，在每一个时点t，我们可以得到一系列的e_t值，它们构成一个向量*v*。用程序可以表示如下：

```
true_N = N
```

```
this_N_est = 1.0
this_E = 0.0
    for e_t in v:
        this_E = this_E * (1-1/this_N_est) + e_t * (1/this_N_est)
        this_N_est = this_N_est*(1-1/true_N) + N * (1/true_N)
```

6.5.7 金融模型的反馈

定量分析的研究人员应该始终铭记于心的是，模型的预测效果市场会很快地回应。也就是说，就算模型能够通过分析数据在市场中赚钱，其效用迟早也会消失。这通常称作：市场的学习能力。

以股票为例，如果模型告诉你市场价格将在未来的某个时点上升，那么你可能会做的就是在现在的时点买入该股票，然后等待未来的时点到来时把手里的股票卖出去以获取价格差的利润。但是试想一下，你在现时点的买入操作其实已经对市场带来影响，告诉市场参与者你预期它价格未来将会走高，这样或会改变其他投资者的决策，从而使得你所预期的信号对股票价格的影响力下降。当然，如果买入量很小，你的买入操作对市场的影响可能微乎其微。但是如果模型一直都表现良好，你的买入量会越来越大，最后你的买入操作可能会对市场产生切实的影响。因为如果你的模型一直预测良好，你的大量买入操作会给其他投资者同样的价格暗示，这会削弱你自己的盈利能力。因此在市场中摸爬滚打其实是在锻炼自己对入仓量的精准把握。

这就是市场的学习过程，每个人都在做出预判，每个人都在看别人如何预判，而市场会做出它最平衡的决定。这样的学习能力所导致的直接后果就是市场的遗忘性，即现存信号对未来市场走势的指导能力很弱。19世纪70年代一次事件的影响会在很短的时间内被市场所预判、理解并消化，我们在2014年是不可能看到任何该事件还可能留存的影响。

确切地说，现今市场每日的记忆能力只有3%。也就是说如果模型的预测值与真实值的相关系数大于3%，说明你的模型很可能已经战胜了市场，你可以从中获利。3%也同样意味着，市场中的可利用信息微乎其微，其绝大多数只能算是噪声。因此，如果模型足够出色，并且交易成本足够小，你才有可能从市场中获利。

因此，机器学习模型中的很多用来衡量模型好坏的标准，比如模型的精确度、准确度等，都不太适用于衡量一个金融模型的优劣。

抛去模型准确度这样的传统测度指标，我们更经常使用的是模型的PnL值，PnL指的是模型所获取的实际收益与实际损失的差值，代表了

模型的日度表现变动情况（注意，是差值而不是比例值），从计算上来看就是今天的表现值减去昨天的表现值。图6-13展示了两个模型的累积PnL值。

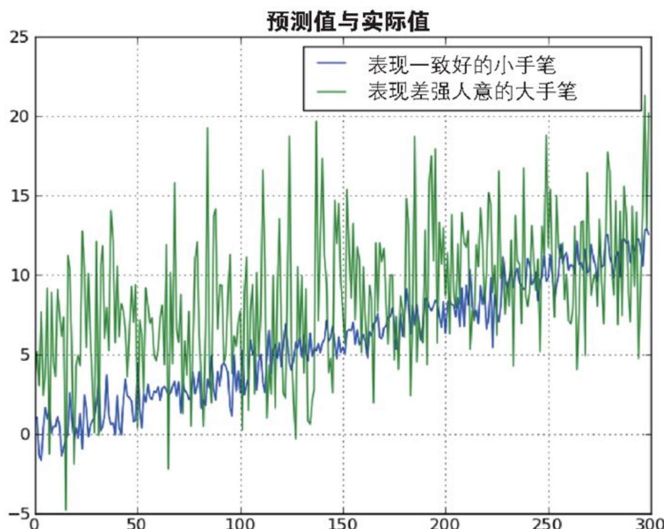


图6-13：两个理论模型的累积PnL值对比图

PnL同样可以应用到其他的很多场合用来评价模型的好坏——基本思想就是画出去均值的预测值以及去均值的实际值的累积和。（去均值也就是将序列的每一个观测值减去其序列均值后得到的序列值。）也就是说，我们想知道所有模型是否一致性地好于那个“最愚蠢的模型”：也就是用均值预测一切的模型。6

> 6从PnL图中我们可以解读出当前使用的模型在样本期内是否一致地优于市场的平均表现。

如果模型的PnL图看起来有一直向右上方前进，则代表的模型一直表现良好。如果走势太参差不齐，你的模型可能有未知的不稳定性。

6.5.8　聊聊回归模型

金融数据中有用的信息含量十分稀少，如果自认为手中掌握的信息可以助你打败市场，还需三思而后行。然而，从形式上倒不妨把金融数据的时间轨迹看作一个连续的、形式未知而复杂的函数。由于函数都可以用泰勒展开表示成一系列成分相加的形式。因此，从形式上，线性模型似乎比较适用于对其建模。

既然线性模型似乎是可行的，那么逻辑回归模型适用与否呢？可以说毫无用处，其对于金融建模的价值基本为0。因为对于金融数据来说，我们需要的是数据的真实值，越精确越好。逻辑回归的二元输出值，对于理解本来信息量就微乎其微的金融数据来说，无非是火上浇油。

线性模型似乎适用于对金融数据的建模，而且对于“线性”二字的理解不能思维定势。“线性”，从统计学意义上来说，是对参数的线性，也就是，你可以对回归变量取平方或者开方项，甚至是交叉乘积项，只要这样项目最后是加总在一起用来建模，都可以称作线性模型。

6.5.9　先验信息量

先验信息量指在模型建立之前，根据历史、行业经验或者主观判断，对于某些参数的分布做出主观性预判，这种预判被数学化之后并整合进了原先的模型之中。在之前指数平滑的例子中，我们假设新数据要比旧数据更加重要，因此权重更大。这就是一个直观的先验信息量。

除此之外，我们可能会做出这样的预判：“模型参数的变动应该是平滑的。”当我们相利用某些时间序列的已有观测值预测未来值时，这样的预判可能是有用的。比如下面的一个时间序列模型：

$$y = F_t = \alpha_0 + \alpha_1 F_{t-1} + \alpha 2 F_{t-2} + \epsilon$$

其中，对于时点t的预测值（$y = F_t$），模型只用到了前两期的历史观测值。当然，如果认为更多的历史数据可能对预测更加有用，模型也可以引入更多的滞后项。滞后项越多，模型引入的自由度（degrees of freedom）越大，我们可以通过引入更强的先验信息量来抵消这些自由度。简言之，先验信息可以降低模型的自由度。

有关模型参数关系的先验信息量的设定（在这个时间序列模型的例子中就是连续相邻数据点之间的关系）可以通过在回归模型估计时，在原数据协方差矩阵上加一个矩阵的形式完成。欲了解更多信息，请参考：http://goo.gl/gJURp6。

6.5.10　一个小例子

分析时间序列时，一个常用的统计量叫作“自回归系数”，可以帮助我们客观地确定合适的滞后项阶数。比如，图6-14就是一个时间序列的自回归系数图，图中的滞后阶数超过了40，达到了100阶。

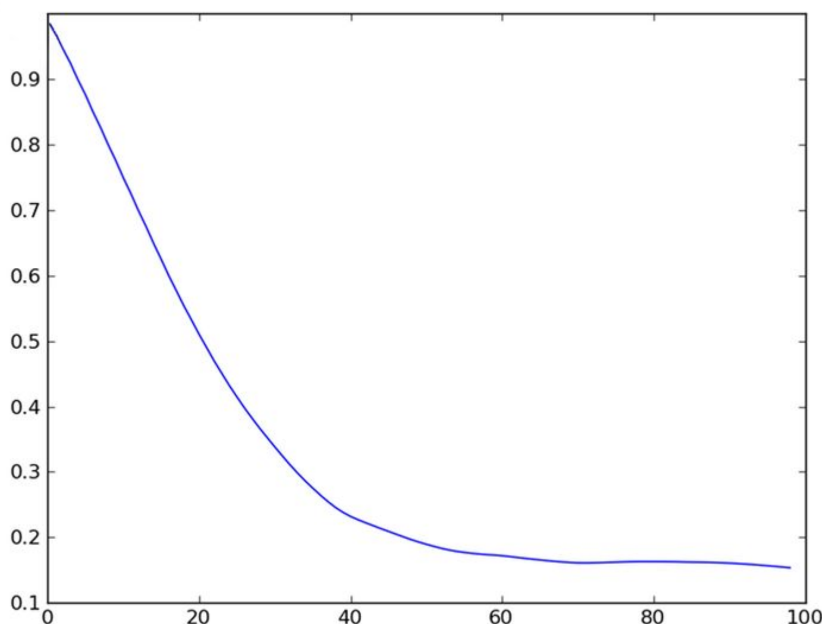


图6-14：某时间序列的100阶自回归系数图

自回归系数的计算很简单。如果要计算1阶自回归系数，首先需要将时间序列整体后移一位（其长度要与原序列相同），原序列和后移序列之间的简单相关系数即代表原序列的一阶自回归系数。

从图中可以看出，在40阶过后序列的自回归系数已经接近0.2，这意味着如果想预测下一个值，使用之前的40个观测值已经足够。然而，40个滞后项对于线性回归模型来说还是太多了。要解决这样的问题，之前说到的先验信息量则大有用武之地。

在模型估计的时候，我们经常会通过最小化一个风险函数来得到最优估计值，这个风险函数代表了模型拟合的效果。7从风险函数的角度来理解先验信息是一个不错的角度：结合了先验信息的风险函数也称作“惩罚函数”，举例来说，对于普通的最小二乘回归来说，一个没有任何先验信息的惩罚函数就是离差平方和项：

> 7比如在回归的例子中，离差平方差就是一个风险函数，通过最小化离差平方和我们可以得到有关模型参数的最优估计。

$$F(\beta) = \sum_i (y_i - x_i\beta)^2 = (y - x\beta)^T (y - x\beta)$$

为了得到最小的F，对F取关于β的一阶偏导数并令其为0，则得到β的唯一最优解：

$$\beta = (x^T x)^{-1} x^T y$$

对于离差平方和项$F(\beta)$，可以加入相应的先验信息量以改善模型的拟合效果。比如说，先验信息表示：模型滞后项中的参数估计的总体规模不能过大，则需要“惩罚”过大的参数估计值。体现在离差平方和中就等同于添加了一个“惩罚项”（$\sum_j \lambda^2 \beta_j^2$）：

$$F_1(\beta) = \frac{1}{N}\sum_i (y_i - x_i\beta)^2 + \sum_j \lambda^2 \beta_j^2 = \frac{1}{N}(y - x\beta)^T(y - x\beta) + (\lambda I\beta)^T(\lambda I\beta)$$

根据相同的微积分求极值的方法，同样可以得到β的最优解为：

$$\beta_1 = (x^T x + N \cdot \lambda^2 I)^{-1} x_T y$$

比较β_1和β可以看到，添加惩罚项对于模型参数估计的影响等同于给数据的协方差矩阵$x^T x$加了一个对角矩阵$N \cdot \lambda^2 I$。细心的读者会发现，这个对角矩阵其实本质上是单位矩阵的常量积。

很明显，先验信息的形式与模型参数的估计值有着直接联系。让我们再添加一个先验信息，并观察参数的估计会产生什么样的变化。假设“参数之间的变动是平滑的”， 也就是说，相邻滞后项的参数估计值之间的差距不应该过大8，那么相应的离差平方和应添加一个新项$\sum_j \mu^2(\beta_j - \beta_{j+1})^2$：

> 8对于时间序列模型，这同样是一个合理的先验信息设定。

$$F_2(\beta) = \frac{1}{\mathbb{N}}\sum_i (y_i - x_i\beta)^2 + \sum_j \lambda^2 \beta_j^2 + \sum_j \mu^2(\beta_j - \beta_{j+1})^2$$
$$= \frac{1}{\mathbb{N}}(y - x\beta)^T(y - x\beta) + \lambda^2\beta^T\beta + \mu^2(I\beta - M\beta)^T(I\beta - M\beta)$$

矩阵M称作移动算子，而I - M在离散微积分中称作离散微分算子（参见http://goo.gl/2D4LeH，以查看有关离散微积分的介绍），该矩阵中除了下三角区域的值为1之外，其他区域的值都为0。$M\beta$的作用是将β的参数顺序整体向前移动一为，最后的位置用0填补，这样的操作对应上式的$\beta_j - \beta_{j+1}$，这也是为什么M称作“移动算子”的原因。

这看起来相当复杂，因此在计算的时候就需要多加小心。$F_2(\beta)$对于β的微分要比之前的复杂很多，这里涉及向量微积分的内容，需要知道：

$$\frac{\partial \mu^T \cdot \mu}{\partial \beta} = 2\frac{\partial \mu^T}{\partial \beta}\mu$$

再求$F_2(\beta)$对于β的偏导数：

$$\begin{aligned}\frac{\partial F_2(\beta)}{\partial \beta} &= \frac{1}{N}\frac{\partial (y-x\beta)^T(y-x\beta)}{\partial \beta} + \lambda^2 \cdot \frac{\partial \beta^T \beta}{\partial \beta} + \mu^2 \cdot \frac{\partial ((I-M)\beta)^T((I-M)\beta)}{\partial \beta} \\ &= \frac{-2}{N}x^T(y-x\beta) + 2\lambda^2 \cdot \beta + 2\mu^2 (I-M)^T(I-M)\beta\end{aligned}$$

令其为0，则可以得到最优解：

$$\beta_2 = (x^T x + N \cdot \lambda^2 I + N \cdot \mu^2 \cdot (I-M)^T(I-M))^{-1}x^T y$$

与β_1的区别之处就在于我们在协方差矩阵$x^T x$上又添加了新的项$N \cdot \mu^2 \cdot (I-M)^t(I-M)$，该项代表了之前设定的“参数变动应该是平滑的”先验信息对最终参数估计的影响。注意，对称矩阵$(I-M)^t(I-M)$的上下两条次对角线上的元素都为1，而主对角线上的元素为2。也就是说，当我们在调整μ时λ也应该做相应调整，因为它们俩之间具有联合变动关系。



先验信息与高阶导数

刚才先验信息的设定只涉及了一阶导数。实际上，你也可以在二阶导数或者更高阶导数上设定先验信息。如果在二阶倒数上设定先验，则相应的$(I-M)$变为$(I-M)^2$。以此类推，高阶导的先验信息对应$(I-M)$的高阶幂值。

那么，到现在为止，我们的模型到底是什么呢？从参数来看，我们需要估计平滑指数γ，另外还需要估计λ和μ：分别对应刚才解释过的$x^T x$和$x^T y$项的移动估计。因此，最后的模型涉及三个需要估计的参数：γ、λ和μ。通常来说，对于需要多大的指数平滑值（γ）我们心里大致有数，但是对于λ和μ的选择则取决于数据本身，以及它们之间

的相互影响。对这些模型参数的最优化估计无非是一件颇需工匠精神的事情：既要保证这些参数的值得到了恰当的优化，又要避免由于过度优化可能导致的模型过拟合问题。

6.6 练习：GetGlue提供的时间戳数据

GetGlue公司为我们提供了一个时间戳数据集以供练习。这些数据与用包含用户签入并评论电视和电影节目的时间戳事件。

原数据（http://bit.ly/1aL8XS0）的时间跨度从2007年到2012年，其中每个用户对应一份数据。数据量占到GetGlue公司用户数据库总量的3%，但即便如此，解压缩之后的数据也有11 GB。

运行下面的R代码可以看到该数据库的前10条信息：

```
# 作者：Jared Lander
require(rjson)
require(plyr)

# 导入数据
dataPath <- "http://getglue-data.s3.amazonaws.com/
getglue_sample.tar.gz"

# 建立连接以解压缩得到的文件
theCon <- gzcon(url(dataPath))

# 读取前10行数据信息
n.rows <- 10
theLines <- readLines(theCon, n=n.rows)

# 检查数据结构
str(theLines)
# 注意，数据向量的第一个元素与其他行有所不同
theLines[1]

# 除了第一个元素，应用fromJSON到向量剩下的所有元素
theRead <- lapply(theLines[-1], fromJSON)

# 转换成数据框的形式
theData <- ldply(theRead, as.data.frame)

# 检查刚才所有操作的最终效果
View(theData)
```

我们给出以下的分析步骤，仅供参考。

1. 先读取1000行的数据并仔细逐条分析。在对这1000个数据有了充分的了解之后，可以考虑读进10万行数据或者上百万行数据做更深入地分析。

2. 做探索性分析，尤其是能够汇总并尝试回答以下几个问题。

- 用户的可能行为有多少种？用户行为的大概分布如何？
- 数据中有多少个不同的用户？9
- 哪10部电影最受欢迎？
- 在2011年有多少部电视和电影节目？

> 9需根据用户的ID信息分辨重复用户。

3. 根据探索的结果，尝试问自己5个新问题。

4. 尝试回答自己的5个新问题。

5. 可视化！如果觉得数据中的某个特征比较有意思，尝试将它们画出来以更好地展示给自己或者别人看。

练习：金融建模

关于金融建模方面，尝试完成以下任务。

1. 获取数据：从雅虎网站的财经板块下载一只至少有8年观测长度的股票每日价格和成交量数据。这是最起码的一步，如果你不知道道怎么做，可以到网上搜索一下。

2. 计算该股票的日度对数收益率。

3. 用同样的方法计算对数成交量的变动值，并与收益率进行比较。

4. 对收益率序列和成交量序列分别建立线性回归模型，并使用至二阶滞后项。用预测值与真实值对比，如果预测值十分准确，没准你能从中赚一笔。如果你还能尝试建立因果模型，对数据进行标准化操作，或者在模型中使用指数平滑，你已经迈入高手行列了。

5. 画出模型的累积PnL图，并与书中的图6-13比较。

第 7 章　从数据到结论

想知道一些公司是如何从数据中提取信息的吗？

本章将由Kaggle公司的William Cukierski以及谷歌的David Huffaker为大家现身说法。

7.1　William Cukierski

William在康奈尔大学获得物理学学士学位，博士毕业于罗格斯大学，研究领域是生物医学工程，研究方向主攻癌症的病理图像研究。在撰写博士论文间隙，William参加了Kaggle上的一些数据分析竞赛，并取得过骄人战绩。现在，他在为Kaggle工作。（下文会有关于Kaggle以及Kaggle竞赛的详细介绍。）

William首先会为大家介绍一些关于数据科学竞赛和数据分析众包业务的内容。另外，他还会详细介绍有关Kaggle竞赛平台的细节。最后的重头戏是关于特征提纯和特征选择的内容。特征提纯是将原始数据中的垃圾信息或者变量剔除，让数据更加“干净”。对于数据分析来说，数据的质量会直接影响模型的预测效果。特征选择是在数据提纯之后，根据数据和研究问题的特点，通过提取、重构、组合等方式构造对于建模有用的预测量。

7.1.1　背景介绍：数据科学竞赛

数据科学竞赛是机器学习领域高手们切磋武艺的常见形式，从其出现至今已经有一段历史了。一般来说，主办方会设计一个较为前沿的机器学习任务，并免费提供分析用的数据。参赛人员/队伍被要求在短时间内（通常是几周或者几个月）设计出相应的机器学习预测模型。至于竞赛的任务则多种多样，有些竞赛要求设计算法，预测人们遭遇车祸的可能性，或者人们喜欢某部电影的可能性等。除了免费提供数据、制定比赛规则和注意事项以外，主办方会公布统一的模型评价标准以及参赛者提交模型的规则等。

比较著名的数据科学竞赛有一年一度的KDD竞赛（知识发现和数据挖掘竞赛），Netflix公司的Netflix Prize竞赛（奖金高达100万美元，前后持续了两年），以及我们待会要详细介绍的Kaggle竞赛。

关于数据科学竞赛有必要在这里做三点声明。首先，作为数据科学生态系统的重要组成部分，数据科学竞赛在一定程度上推动了数据科学的发展。作为数据科学家，应该密切关注数据科学竞赛的最新动态。

其次，数据科学竞赛的内容和形式在一定程度上定义了数据科学本身。竞赛的不断发展给数据科学的应用建立了很好的内容库，同时也在不断地更新着数据科学的定义。当然，这并不代表数据科学竞赛的内容直接定义了数据科学的内容，只是说，在一定程度上，竞赛的内容为了解和推广数据科学本身提供了鲜活的材料：数据科学到底是在处理什么样的问题，数据到底长什么样子，数据科学忽视了哪些因素等。

第三点，数据科学竞赛的结果往往都会以个人或者团队排名的形式公布，排名靠前的往往被视为顶尖的数据科学家。然而事实上，很多优秀的数据科学家，尤其是女性，并没有参加任何形式的数据科学竞赛。这代表她们不可能获得任何排名，但她们却同样优秀。因此，我们应该客观理性地看待竞赛的排名。



数据科学竞赛：到底谁在参与竞赛

各种各样的数据科学竞赛看来十分正规、程式化并且有条不紊。主办方已经准备好了数据、模型的评价标准以及需要解决的问题，甚至对于建模之后如何可视化和报告都有详细的指导。比如说Kaggle公司的数据科学家就是在做这些事情：找有意义的数据，做好基本的清理工作，尝试建模，寻找合理的评价标准，构思有意义的分析问题等。这些工作本身应该成为数据科学的一部分，应该成为数据科学家必备的技能。而主办方似乎把这些事都办得妥妥的，数据科学竞赛中“数据科学”的味道已经被大大地弱化了。

7.1.2　背景介绍：众包模式

这里讨论以下“众包模式”的概念，总体来说有两种众包模型。第一种叫作分布式众包模式，其典型代表是维基百科。这种众包模式适合任务相对简单而贡献来源分布较广的情形。在维基百科上，来自世界各个角落的人都可以贡献内容，由志愿者组成的幕后团队负责管理和控制内容的质量。维基百科制定了一套完整的内容质量管理系统。这种

众包模式的结果是集众人的知识和力量搭建了一个内容可信度极高的在线百科全书。

另外一种众包模型解决的是更加专业、困难和具体的问题。Kaggle、DARPA、InnoCentive等很多公司都在从事这样的众包业务。这些公司将某个较为专业的问题公布于众，往往只有很少的一部分人具备解决该问题的能力，因此竞争只在这一小群人之中展开。获胜者往往可以得到可观的现金回报，当然还有随之而来的荣誉和业界的肯定。

后者这样的众包模型，其最大的短板是是参与者数量太少。究其原因，首先很多竞赛往往没有或者缺乏合理公正的模型评价标准。在一些竞赛中，对于输赢的评价标准往往不够客观，掺杂着个人主观因素。比如说，你的模型是好是坏单单由评委说了算，而评委的喜好也各不相同。其次，参赛者的沉没成本相当高，因为只有获得名次才有奖励和荣誉，只有金字塔顶的人才能拔得头筹，往往还需要一定的运气。这些都直接导致了数据科学竞赛的参与者寥寥。

组织问题也同样妨碍了数据科学竞赛的健康发展。有些随意设计、组织混乱的竞赛往往让参赛者分析一些毫无意义、枯燥无比的数据。这些组织者往往认为无论出什么题、给多少奖金，参赛者都会摩拳擦掌，跃跃欲试。这些无意义的竞赛严重打消了数据科学家们的积极性。有些竞赛的题目要么过于庞大，让人找不着北，要么过于细碎，让人兴致全无。

既然已经认识到了这么多妨碍众包模式发展的因素，我们可以预期一个好的众包竞赛首先应该精心设计竞赛题目，做到有趣、可行又有实际的商业意义。模型的评价标准应该足够透明和客观。为了充分提起参赛者的兴趣，奖金当然越高越好，不同名次奖金的分配和颁发，也应该合理而透明。

众包竞赛的发展其实已经有了几百年的历史，下面是一些影响力较大的竞赛，让我们回顾一下。

- 公元1714年，英国皇家海军因为地球经度的测量问题1伤透了脑筋，于是拿出相当于现在600万美元的奖金征求能够解决该问题的人。一位名不经传的家具工John Harrison巧妙地利用钟表原理制造了一种叫作“经线仪”的工具解决了这个问题。

- 2002年，福克斯电视网络公司推出大型电视选秀节目“美国偶像”，意在海选出下一代流行歌王/歌后。该节目开创了电视选秀节目的先河，选手在一轮又一轮的淘汰赛中比试歌喉，最终

获胜者可得到巨额奖金。

- X-prize公司（http://www.xprize.org/）是一家专注于大型科学竞赛的推广公司。这些竞赛的内容往往涉及与人类生存息息相关的重大科学问题，一旦解决，可以为传统产业注入新生血液，甚至会开创一个全新的产业。因此，竞赛的另一大特色就是：奖金奇高。其中的Ansairi X-prize是一个空间科学竞赛，其奖金高达1000万美元。这样的问题往往需要科学界的顶级专家，花费大量的时间才能解决。因此，这样的众包竞赛由于门槛过高，参与人数少，往往要持续很长的时间。但是，这对于相关高精尖领域科学的发展还是明显有推波助澜的作用。

1如何确定船舶在东西方向的具体位置。

关于众包和土耳其机器人

众包和土耳其机器人这两个名词在数据科学领域悄然地出现，并且越来越多地引起人们的关注和重视。

虽然“众包”一词的正式出现是在2006年，但其概念本身却并不新鲜：针对一个问题，多人一起各自独立提出解决方案，综合一起就会得到一个更佳的方案。Jame Suriowiecki的*The Wisdom of Crowds*（《群体的智慧》）（Anchor，2004）一书详细讲述了众包的理论含义，其核心观点是：三个臭皮匠顶过一个诸葛亮。当然，群体智慧发挥作用需要一定的条件，其中首要的是“独立性”，也就是群体中每个个体应该独立思考并提出自己的解决方案，而不是采取群体讨论的方式给出小组答案。当然，并不是所有的问题都适合交给三个“臭皮匠”来解决，有些问题还是需要“诸葛亮”（领域专家）的帮助。

亚马逊推出的“土耳其机器人”业务是线上众包服务的代表。比如，表情识别对于人眼来说是个很简单的问题，看一眼就知道表情是“开心”还是“难过”2，但这对于机器学习来说是个极富挑战性的问题。对于表情识别来说，监督性学习的目的就是给定一批图像的标签（是“开心”或“难过”），通过算法学习和识别新图像中的表情是“开心”还是“难过”。问题是，如何事先得到一批已经做好标签的图像呢？这就是“土耳其机

器人”业务的出发点，因为这样的贴标签任务只能通过人工手动完成，“土耳其机器人”业务就是为用户提供图像贴签服务。网络上的每一个人都可以注册成为亚马逊的“土耳其机器人”，只要你有闲暇时间就可以选择坐下来给一些图像贴标签，正好还可以赚点外快。因为这样的任务没有太多技术含量，也比较枯燥乏味，固冠名为“机器人”。当然，为了防止“机器人”们闭着眼睛瞎填，亚马逊设计了严格的质量控制系统。一旦发现作弊，你就永远失去了做“机器人”赚外快的资格。

“土耳其机器人”看似是人的体力劳动，但其终极目标是结合机器学习算法和少数人的劳动成果，最终减少了大多数人的重复性劳动量。因为那些被贴好标签的数据会被各种各样的监督性机器学习算法用来预测大量的未被贴标签的数据。

> 2当然，也可能没表情。

7.2 Kaggle模式

数据科学家走在通往无所不知的大路上，走到尽头才发现，自己一无所知。

——Will Cukierski

Kaggle的口号是“让数据科学成为一项竞技赛事”。Kaggle是其他公司与数据科学家之间的联络人。比如说，一家公司想将数据分析任务众包给数据科学家，Kaggle会揽下该活并收取一笔佣金。因为Kaggle提供的数据科学竞赛平台吸引了来自世界各个角落的数据科学高手，这些众包的任务只需要交给自家平台上的数据科学家即可。

当然，Kaggle公司内部也是藏龙卧虎之地，许多顶尖的数据科学家都在Kaggle工作，Will就是其中的一员。提供分析任务和数据，并付钱给Kaggle的是需要数据分析众包的公司，而解决这些任务的是Kaggle平台吸引的来自世界各地的数据科学家。Kaggle的数据科学平台，任何人都可以注册参赛，基本没有门槛限制。下面我们会分别从参赛者和需求方公司的角度分析一下Kaggle的业务模式。

7.2.1 Kaggle的参赛者

在Kaggle的数据竞赛中，参赛者可以拿到一个训练数据集和一个测试数据集3。竞赛的题目大多数是有关监督性机器学习的，也就是说在训练数据集中，x和y的值都是给定的，用来训练监督性学习算法。测试数据集中只会提供x，不提供y。参赛者在训练数据集上训练模型，并将模型应用于测试数据集，得到预测的y值。如果参赛者对模型的效果满意，可以在Kaggle平台上提交预测的y值。Kaggle的后台系统会根据事先设定好的模型评价标准，比较参赛者提供的y值和真实y值的差距，并给出参赛者的实际得分值。在这个过程当中，参赛者不需要上传代码而只需要上传模型的预测值，因此Kaggle并不在意你用什么分析软件。当然，最终的获胜者需要提供代码。测试数据集中的x值虽然不包含真实的y值信息，却也十分重要，它无论在变量个数、含义和格式方面都和训练数据集中的x值完全一样。为了保证比赛绝对公正，尽量降低作弊的可能性，在竞赛截止日期之后，Kaggle会用第三个数据集4测试参赛者最终的模型效果，并据此判定最终的排名。这个数据集无论是x值还是y值，参赛者都无从得知，从而可以最大程度上保证数据的独立性和模型效果的客观性。5

> 3需要同意一个数据使用协议。

> 4不同于参赛者拿到的数据集，也不是竞赛期间Kaggle后台评分用的数据集。

> 5Kaggle在竞赛期间会设立排行榜，在排行榜上的获胜者并不是最终的获胜者。因为排行榜的结果来自于提供给参赛者的测试数据集，最后的独立数据测试结果决定了最终的冠军归属。

为了防止参赛者频繁地提交，Kaggle的竞赛一般限制参赛者一天最多只能提交5次，每个竞赛持续几周到几个月不等，甚至更长。每次提交，Kaggle可以迅速计算出模型的效果并通过排行榜的形式反馈给参赛者。排行榜上可以看到每个参赛者的名字和最后提交的模型的预测误差。如果一个赛事有足够多的参赛者，可以看到图7-1中那样“前赴后继”的壮观场景。只要一个参赛者的模型有了细微的改进，别的队伍会迅速跟进。比赛的效果很明显，每个参赛者的模型效果在比赛期间都有或多或少的改进，最后的冠军凭借0.791的模型准确度拔得头筹。可以想象，如果没有这样一个竞赛平台而只是单干的话，数据科学家不断改进模型的本能和冲动很难被激发出来。

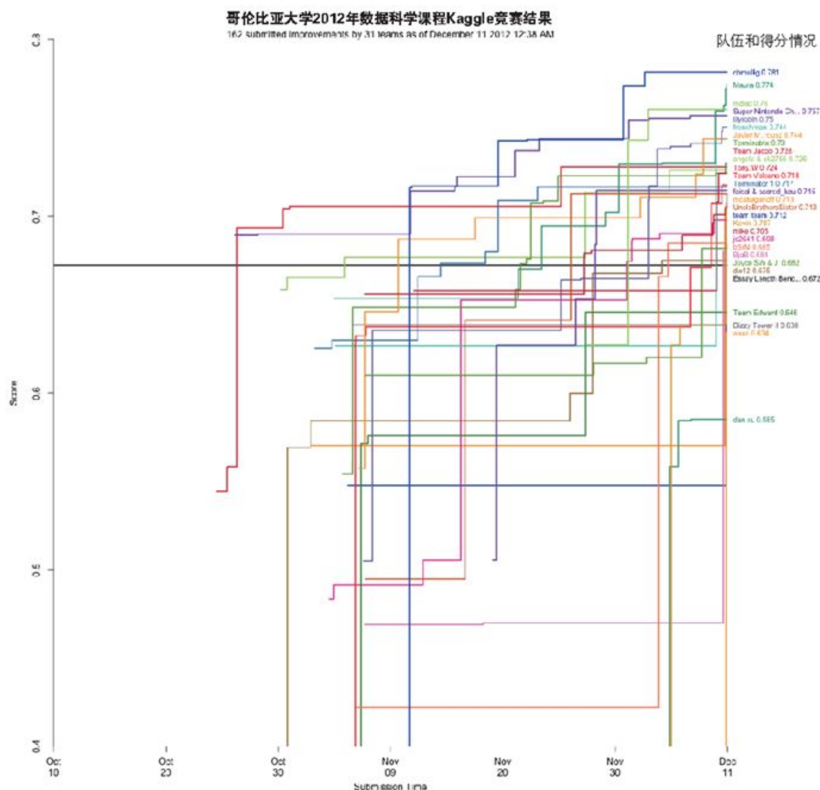


图7-1：该图出自Chris Mulligan， 他是Rachel班里的学生。该图很好地描述了每个参赛个人/队伍在比赛期间，模型的进化情况

这样“前赴后继”改进模型的场景固然会带来预测效果更好的模型，但同样也会带来更加复杂的模型。为了不断改进模型，参赛者会使出浑身解数。这也是为什么一般的竞赛至多只会持续数周或者数月的时间，因为模型的预测精度总有上限，在进化一段时间之后就会进入瓶颈期，要想提高模型的精度，参赛者会把模型打造得越来越复杂。以Netflix Prize为例，这个为期两年的竞赛，其最后的获胜模型由于过于复杂，Netflix公司都没有在实际产品开发中应用该模型。**6**

> 6当然，奖金还是如数发给了获胜队伍。

7.2.2　Kaggle的客户

Kaggle的客户是一些需要数据分析的公司，那么这些公司为什么愿意

付钱给Kaggle呢？原因很简单，因为Kaggle的平台上有许多活跃度很高的数据科学家们。一些公司存储了海量的数据，并且每天都有大量的新数据产生，但苦于没有能够分析这些数据的科学家们，Kaggle作为中间人恰好填补了这个空白。个别公司会自己设计数据竞赛吸引数据科学家们。Kaggle模式的创新之处就在于，它凭借自身的平台优势，说服客户捧出他们手中宝贵的数据。客户往往会欣然接受，因为Kaggle平台上的数据科学家们总会帮助解决令他们头疼的数据分析问题。

迄今为止，Kaggle竞赛已经为业界带来了不少优秀的数据分析解决方案。Allstate是一家汽车保险公司，虽然他们公司内部已经有一只素质良好的精算师团队，他们还是把数据放在了Kaggle上寻求更优秀的解决方案。Allstate给出的分析任务是：给定汽车驾驶员的属性信息，预测其发生车祸的可能性。Kaggle上一共有202个数据科学家参与了这项竞赛，竞赛的结果令人瞠目结舌。冠军的模型效果，以正则化的Gini系数为标准，将原本Allstate公司的模型效果提升了271%（详情可见http://www.kaggle.com/solutions/casestudies/allstate）。这样的例子还有很多，其中某公司在Kaggle上发布了奖金额仅为1000美元的竞赛项目，最后从中的获益，据保守估计，超过数十万美元。

> 这样公平吗？
>
> 与前面的Allstate例子类似，这些公司内部已经有了一支素质良好的数据分析团队，如果这些公司还在Kaggle上寻求解决方案，对公司内部的数据科学家固然不太公平。如果Kaggle竞赛的获胜模型明显优司内部开发的模型，这些员工很可能会被扫地出门。
>
> 竞赛中只有排名前三（甚至第一名）才有可能拿到奖金，这意味着绝大多数人都在免费地拼命干活，而公司往往会从Kaggle的获胜模型中获益颇丰，这对绝大多数参赛的数据科学家们公平吗？
>
> Kaggle会对这些公司收取佣金，有些公司所公布的奖金额也十分慷慨，至于参不参赛，数据科学家有着绝对的自主权。
>
> 这些因素加起来，Kaggle对于三方来说似乎都是公平的。真是这样吗？
>
> 其实得益较多的应该是Kaggle的客户，数据科学家往

往意识不到他们所开发的模型以及花去的时间价值要远远大于奖金额。除非对某个竞赛题目或者数据由衷得感兴趣，否则在从事这个基本上等同于零回报的数据竞赛之前，数据科学家们还是要三思而后行。

最近Facebook在Kaggle上发布了一项数据科学竞赛，获胜者可以获得Facebook的面试机会。最后的参赛人数达到了422人。我们认为这极大地方便了Facebook招募优秀的数据科学家，但是Cathy却觉得这很可能会把数据工作者们引入歧途：他们专注于分析数据和解决问题，却忽视了公司政策和文化底蕴，这对于公司和员工来说同样重要。‘

Kaggle的作业自动评分模型

还记得本书开头提到的，在哥伦比亚大学开设的数据科学课吗？作为该课期末考核的一部分，学生被要求建立一个作业自动评分系统。考核方式完全参考了Kaggle竞赛的模式，学生之间也可以组队完成任务。下文将详细介绍该作业自动评分系统的内容，数据可以从https://inclass.kaggle.com获得。

训练数据集是一批已经被老师批改完的作业，*y*值是作业的最终得分，*x*值是作业的相应特征变量。测试集只有*x*值，会被用来评判模型的最终预测效果。

竞赛中一共用到了5组不同的作业数据集，每个作业数据集都生成自一个单独的提示语（prompt）。每篇作业的总字数在150到550之间。某些作业可能跟原信息源相关，而有些则与信息源无关。完成每篇作业的学生其成绩都在7到10之间。每篇作业都由两名老师独立手工批改并打分。除了这些共同特征变量，每组作业还有相应的特色变量。数据所特意生成的特异性旨在测试评分系统引擎的灵活程度。具体来说，数据集会包含以下几列：

- id
  作业的独立编号
- 1~5
  作业来自的组号。一共有5组不同的作业。
- essay
  用ASCII码表示的作业的具体内容
- rater1

第一个老师的打分
- rater2
  第二个老师的得分
- grade
  综合分数

7.3 思维实验：关于作业自动评分系统

我们在课堂上问过学生们对作业自动评分系统持什么态度，他们是否愿意该系统去批改他们写的作业，以及这个系统可能会带来什么样的好处和坏处，等等。下面是一些学生的回答。

- 自动评分系统更加客观。

  已经有研究表明，医生在两个月前后对同一张X光片的诊断会截然不同。这很正常，因为人的思维具有不一致性。即便你认为自己毫无偏颇，然而事实就是事实。从这一点上来看，机器给出的判断往往比人的判断要更加客观，前后一致。机器学习甚至已经被广泛应用在了癌症研究上，要知道这是一项高风险的任务，但是机器似乎并不比人做得差。

- 机器过于程式化，因此缺乏创新能力？

  机器评分可以使整个批改任务标准化、流水线化。人们通常喜欢标准化的东西，这样虽然保守，但有更好的一致性。比如说汽车，相比于手工打造的汽车，人们会更加信任流水线生产的汽车，因为它更加安全可靠。但问题是，并非所有的任务都适合程式化。至于批改作业到底适不适合标准化，这还是要考量作业的内容、形式、难度等诸多因素。

- 考试的目的就只是要学生们写出好论文吗？

  这里假设考试的形式是一篇小型论文，通常老师会给出详细的写作大纲和评分标准。如果严格地按照大纲和标准来说，得高分并不是一件难事。那些培训机构甚至专门著书教导学生们如何应试。那么这种应试技巧有没有办法翻译成机器语言呢？一个可能的办法就是设计机器学习算法，通过老师给出的大纲和标准，自动化生成论文。如果真是这样，教育过程就变成了机器之间的竞技——是学生和老师的算法之间进行博弈。而在这场战役中，我们认为学生的算法获胜可能性更大。

领域知识与机器学习算法

领域知识和机器学习并非水火不容的关系。恰恰相反，数据科学问题的解决离不开二者中的任何一个。然而Kaggle的主席Jeremy Howard先生在2012年12月份的《新科学家》杂志接受了Peter Aldhous的专访时说道：“专家的知识基本上毫无用处，我们根本不需要。”这下可把专家们惹毛了。下面是那次专访的实录，我们来听听他到底说了些什么。

PA：Kaggle比赛的获胜者和表现平平者有何不同？

JH： 从技术上来看，Kaggle比赛获胜的关键是将最有用的信息交给模型。因此，你必须决定从原始数据中抽取多少信息、何种信息、以什么样的方式交给模型。Kaggle比赛的获胜者一般都具有强烈的好奇心和创新能力，对于同样的问题，他们擅于从许多不同的角度解析、分拆和组合问题。一些像随机森林这样的算法并不在乎你有多少想法，只要你的想象力足够丰富，只管喂给模型就好了。随机森林会自动使用那些更加有用的信息。

PA： 这听起来与传统预测模型的建立过程完全不同，通常领域专家会扮演十分重要的角色，决定采用或者不采用哪些特征变量。这些专家对Kaggle有何评论？

JH：真要我说的话，可能会得罪很多人。我想说的是，这些专家积累几十年的所谓“领域知识”基本上毫无用处，我们在建模时根本不需要领域专家的帮助。专家们所使用的看似花哨的模型，其效果比单纯的机器学习算法要差很多。当然，这个观点很多人会驳斥，因为在过去几十年内，专家的意见总是主导着人群的主流意见。但是他们总是花很长的时间讨论一个变量是否有用，在细枝末节上钻牛角尖，我真的认为是在浪费时间。

PA： 那你觉得专家的知识还能在哪派上用场？

JH： 也许在建模初期会有用，因为专家们通常比较擅于发问，他们可以帮助我们提出更好的问题。但接

下来就是数据科学家的战场，基本跟他们没有什么关系了。

PA：Kaggle上充斥着大量的数据驱动的、黑匣子似的算法，即便是建模者自己也不知道模型到底如何解释。你觉得这种方法是否存在弊端？

JH： 有些人认为，使用算法解决问题的弊端在于，即使最后得到了答案，也不代表着你对问题有更深入的认识。但模型的解释性果真如此重要吗？我觉得不然。算法会告诉你哪些变量重要，哪些变量不重要，这已经足够了。你或许会问，这些变量为什么重要，而为什么这些变量不重要。我觉得这些问题就很无趣了：既然你已经得到了一个预测效果相当好的模型，就不必再对模型的可解释性吹毛求疵了，这完全没有什么必要。

7.4 特征选择

在数据建模中，数据科学家从数据中选取哪些特征变量，以怎样的形式加入模型，这个过程叫作特征选择。

特征选择对模型的效果往往有着直接影响。Will在被Kaggle招安之前，屡次在各个Kaggle数据科学竞赛中斩获佳绩（这也是为什么Kaggle会招他的原因），因此他对怎样建立一个有效的模型有着充分的发言权。特征选择不仅可以帮助你在竞赛中取得好成绩，更重要的是，它直接的影响模型或者算法的效力。原始数据固然隐藏了所有的信息，但只有通过特征选择攫取出来的信息才能为模型所用。

原始数据中存在大量的冗余信息，比如很多相关性很大的变量。如果一股脑得全放进模型，必然不会有很好的效果。这些信息需要经过挑选、转化或者组合，才能为模型所用。比如说，对变量取对数，将连续性变量离散化为二元变量都是基本的转化形式，它们往往会在一定程度上提高模型的预测能力。



术语解释：特征、解释变量和预测变量

> 不同的学术领域会用不同的术语表述相同的对象。统计学家喜欢用“解释变量”“自变量”或者“预测变量”等表示模型的输入变量。而计算科学家往往统称为“特征”。

> 特征提纯和选择对于机器学习的重要性经常被人忽视，但却是其中最为重要的一环。选择更好的特征变量往往可以得到效果更佳的模型。

——Will Cukierski

> 我们使用的算法与人无异，我们只是数据比他们多而已。

——Peter Novig，谷歌研究总监

Will觉得Peter Novig所提到的更多的数据，应该指的是更好的特征。因为更多的数据有时候并不会带来额外的信息。（比如说，你关心掷骰子过程中2出现的概率，那么掷骰1000次骰子和10 000次骰子所得到的结果差别并不大，因为很可能当你掷第500次的时候，2出现的概率已经稳定在了1/6左右，并且不再有明显的改变。因此更多的数据在这里并不代表更多的信息。）谷歌公司所要解决的数据问题往往都纷繁复杂，因此搜集更多的数据往往能够带来更多更有用的特征变量，这也许是Peter Novig真正想表达的意思。

然而，过多的特征也并不见得就绝对是件好事。比如说，当数据中特征变量的个数超过观测值的个数时，会产生稀疏性的问题。这会导致很多模型的参数估计失效。因此，过多的变量也并非是好事。有时候，太大的数据还会带来额外的存储问题和计算问题：因为数据太大，一个计算机的内存很难一下子装进如此多的数据。因此数据必须被分割，存储在不同的计算机上。这是数据科学中一个令人头疼的问题，它往往会带来不小的负面作用。

然而，总体来说，要想得到好的预测模型，特征选择是不可逾越的一步。

7.4.1　例子：留住用户

假设要你设计了一款叫作“追龙”（Chasing Dragons）的手机游戏（程

序图标如图7-2所示），用户以交月费的形式使用该程序。那么很明显，用户数量越多，你赚的钱就越多。然而事实上，只有10%的新用户会在下一个月度选择续订。因此如何吸引更多的新用户，以及如何留住老用户是你需要考虑的两个最重要的问题。通常来看，招揽新用户的成本要大于留住老用户的成本。但是我们这里只关心其中一个问题，就是如何更有效地留住用户。应用机器学习模型，我们可以根据新用户的特征预测该用户下一个月是否会继续续订你开发的这款游戏。模型本身也会有助于你理解产品特征、用户行为等对用户黏性的影响。但我们更加关心是模型的预测效果到底好不好。如果模型的预测效果很好，对于那些可能会退订的用户，你可以采取相应的激励措施（比如一个月的免费使用权）想方设法留住该用户。

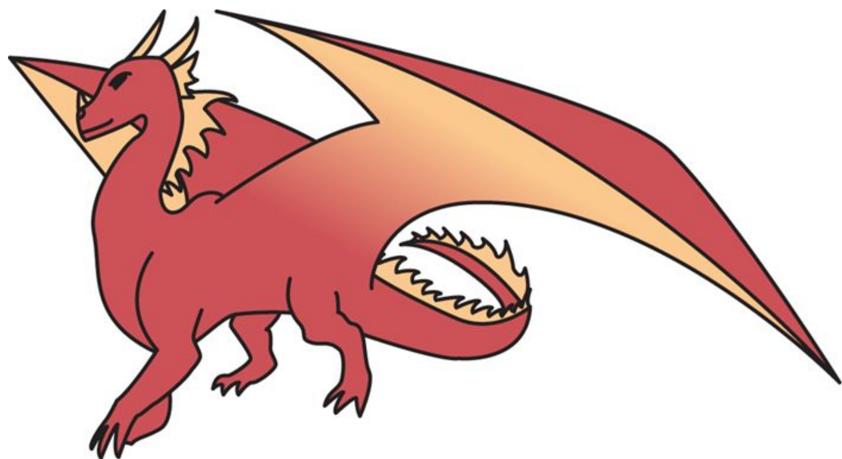


图7-2：你设计的手机游戏程序："追龙"

什么模型比较适合呢？第4章中，我们说到了逻辑回归模型主要适用于预测二元变量，因此可以先试试逻辑回归模型。模型的输出值是用户在下一个月续订的概率值。（统计学中的生存模型在这里可能更加适用。）在用户下载程序的那一刻起，数据的收集工作就开始了。在第一个月的30天时间里，要尽可能详细地记录每位用户的使用行为习惯等，并根据用户的行为数据预测其在下一个月续订的概率值。数据的形式可以是时间戳的形式，比如：用户A在早晨5：22分打开了游戏程序，5：23分杀死了一只龙，在5：24分拿到了24分，5：25分游戏自动生成了一条有关香体露的广告，等等。严格来说，用户的每一个动作都可以也应该被详细地记录下来。

活跃用户和非活跃用户之间的行为差别巨大，前者会产生大量的行为数据，而后者可能只有零星的记录。在数据分析前，数据需要被转化

为可分析的矩阵形式，行代表用户，列代表用户的行为特征变量。在数据收集和整理阶段，原则上应该完整记录下用户的每个行为特征，不做任何删减，这个阶段称作“特征生成”，而不是“特征选择”。但是，应该构建哪些行为特征变量可是需要团队紧密协作的，团队成员应该坐下来进行激烈的头脑风暴。因为这对之后的模型效果有着直接的影响。下面是一些可能的变量，可作参考：

- 第一个月内用户访问游戏程序的总次数；
- 用户第一次使用游戏程序的持续时间；
- 第一个月内，用户每天的游戏得分（也可以是30个变量，每个变量代表一天的得分）；
- 第一个月的游戏总得分；
- 用户是否填写了完整的个人信息（二元变量，1代表已填写，0代表未填写）；
- 用户年龄；
- 用户性别；
- 用户手机的屏幕尺寸。

这样的特征变量越多越好，即便有些变量之间的差异可能很小，或者相关性很强，也没有关系。

特征生成与特征提取

刚才所说的“特征生成”过程也称作“特征提取”。这个过程既是一门科学，又是一门艺术。做特征提取时，有领域专家从旁指导固然是好，若没有的话，充分发挥你的想象力也会收到不错的效果。

只要想象力足够丰富，你可以得到成千上万的特征变量。如此多的特征变量对于像问卷调查这样的学科是难以想象和企及的。有经验的人会知道，一篇问卷的问题一般不过几十个，受访者能够仔细真实回答的也不过20个。

如果特征变量过多，难免会有一些是不太有用的。有些你能够想到的变量，在数据里也并不一定能够找到。因此，在头脑风暴这些特征变量的时候，可能会遇到以下几种情形。

- 有些变量可能与问题十分相关，对建模也十分有用，但是在数据中并不一定存在。

用户的很多信息是我们无法记录的，比方说，用户的闲暇时间；用户在玩哪些别的游戏，使用哪些别的程序；该用户是否失业下岗；该用户是否失眠；用户是否容易对游戏上瘾；用户是否对龙比较反感，以至于会做恶梦。这样变量固然对我们理解用户的行为，以及后期的建模都十分有益，但问题是这些数据大多是很难直接获取的。当然，有些变量我们可以通过间接的方式获得。比方说，如果用户玩该款游戏的时间总是在凌晨或者深夜，那么很可能该用户会有失眠症；不过，也有可能他们正在上夜班。

- 变量与问题本身有关，并且已经通过程序本身追踪和存储。

  然而需要注意的是，有些可以追踪和存储的变量信息，你很难确定它是否对后期的建模有任何用处。这就是为什么在“特征生成”和“提取”的过程中要尽量多的生成变量。变量的选择工作需要交给后期的“特征选择”工程来做。

- 变量与问题有关，程序可以追踪和存储该变量，但实际上却没有做到。

  比方说，一个可能重要的变量是“用户是否上传了头像”。然而，你或者你的团队却认为这个变量并不重要，或者你们根本没有讨论过这个变量。这都有可能，任何人或者团队都不会做得面面俱到。这样的变量对建模有很大的帮助，也可以被轻松地记录和保存下来，但最后却没有。然而，对产品事先做好“可用性研究”可以有效地避免忽视掉重要的解释变量（本章的后半部分中，David Huffaker会介绍有关“可用性研究”的内容）。因为在“可用性研究中”，我们会更多地从用户角度考虑产品设计的方方面面，可以帮助确认需要追踪用户的哪些产品使用行为。

- 变量可能毫无用处，但还是被记录了下来。

这太常见不过了，很多记录下来的特征变量都被证明是毫无用处的。但这无伤大雅，因为后期的“特征选择”会过滤掉这些变量，而保留下那些真正重要的变量。

- 变量可能毫无用处，也无从追踪和保存。

  对于这样的情况就不需要无病呻吟了，既然没有用处，则根本不要追踪和保存，因为你根本不需要它。

现在该回到刚才逻辑回归模型的问题了。用$c_i = 1$表示用户在下一个月的某个时间选择了续订“追龙”游戏。到底是下一个月，下半个月还是下一周，你的团队应该讨论决定，但在建模初始，你和团队不需要太在意这些。等模型的效果稳定之后，可以再商讨这些细节。

通过前章的学习，我们知道逻辑回归的模型形式为：

$$\text{logit}(P(c_i = 1|x_i) = \alpha + \beta^T \cdot x_i$$

但你真的会把刚才生成的成千上万的特征变量一股脑全扔给模型吗？即便你这样做理论上没有问题，模型还是会照样运行，但其预测效果想必不会太好，而且运行起来十分缓慢。“特征选择”就是用来从变量集中精选出最为重要的一些变量，既可以提高模型的预测效果，也可以提升模型的运行速度。

关于“特征选择”，Will强烈推荐大家读一读Isabelle Guyon于2003年撰写的一篇论文“An Introduction to Variable and Feature Selection”（“变量与特征选择导论”，参见http://goo.gl/3dz8Ar）。该文从提高模型预测能力的角度详细讨论了如何更好地选择特征变量。变量选择与变量生成以及变量排序有着本质的不同，Isabelle在文中将主流的变量选择方法分为了三类，过滤型，打包型和内嵌型。下文将分别介绍这三种方法。

7.4.2 过滤型

过滤型是指根据特征变量与因变量之间的某个统计量的值将特征变量从大到小排列出来，再根据排序的顺序过滤掉一批变量。比如，可以用简单相关系数作为统计量。该方法的特点是每个变量都是独立过滤的，不考虑变量间的相互作用。因为其简单有效，通常会用在变量选择的第一步。

过滤型方法的好处是其容易计算，实施起来比较简单。但是，由于其并未考虑变量间的相关信息，可能会有一些副作用。Isabelle在文中解释说，被过滤掉的两个“不重要”变量单独来看可能确实都不重要，但是可能放在一起可能会非常重要。变量之间的相互作用往往会将某些不重要的变量联系并组合成一个比较重要的特征。

当然，统计量的类型并不限于简单相关系数。对于线性回归来说，一种较为常见的过滤器可以基于模型参数估计的p值或者模型拟合的R方。其操作方法是，用每一个特征变量单独建立回归模型并计算该变量参数估计的p值或者该模型的R方。最后，根据p值大小7或者R方的大小8对变量排序（在接下来的“什么选择标准合适”中还有详细介绍）。

> 7越小说明该变量越重要。

> 8越大说明模型的解释能力越强，该变量越重要。

7.4.3 包装型

包装型方法是从所有的变量集中（假设变量总数为n）找到一个最优子集并打包给模型使用。通常子集的大小是一个固定值k。学过排列组合的同学应该知道，从n个变量中选取k个变量的可能方法有$\binom{n}{k}$种，因此k的选择至关重要。

在应用包装型的变量选择时，有两方面细节需要考虑：其一是选用什么样的算法选择最优子集，其二是选用什么样的“选择标准”衡量一个变量或者变量子集的优劣。

什么算法合适

我们先讨论一种最为常见的变量选择算法，叫作分步回归。对于回归模型，该方法的特点是，变量以一种系统性的方式被放入模型（或者从模型中移除）中，同时用类似R方这样的选择标准记录某个变量被放入模型（或者从模型中移除）时对整个模型效果的影响。通常的分步方式有三种：向前搜索法、向后消除法和双向搜索法。

- 向前搜索法

  向前搜索是指先建立一个没有任何变量的回归模型9，每一步都从变量集合中选择一个变量加入到模型中，选择的标准是看哪

个变量可以最大程度地提高模型的拟合或者预测效果。变量被加入到模型之后就不会再从模型中移除，下一步总是从剩下的变量集合中找出一个最佳变量加入到模型当中。每次只添加一个变量，直到所有剩下的变量都不能提升模型时候，模型的向前搜索即停止。

- 向后消除法

  从搜索的方向上来说，向后相除法与向前搜索法正好相反。刚开始需建立一个包含所有变量的回归模型（称作全模型），然后每次从变量集合中移除一个变量，移除该变量的效果基于其可以最大程度上提高模型的拟合或者预测效果。变量被移除后就不会再返回模型中去，直到移除任何一个变量都不会提高模型的拟合或者预测效果时，模型的向后消除即停止。

- 双向搜索法

  由于向前和向后消除法的特点是每次只加入或者移除一个变量，而一旦变量被加入就不会再被移除，或者一旦被移除就再也不会返回模型当中。这样的算法对于可能过于绝对，双向搜索法就可以用来解决这个问题。

> 9意思是，只有一个截距项的线性模型。

什么选择标准合适

选择变量的标准有很多，作为一个数据科学家，你还需要在众多选择标准中做出选择。

虽然每一个选择标准都有相应的理论可供参考，但更多的时候，选择起来还是相当主观的。一种可行的选择方法叫作“试错法”：尝试几个不同的选择标准，看模型在哪个标准下表现的更加稳健。不同的选择标准会选出截然不同的模型变量，因此你应该对主流的选择标准有所了解。

- ***R*方**

  R方的定义公式为：

$$R^2 = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y}_i)^2}$$

线性模型的R方大小代表了数据中有多少的信息可以被该模型拟合。

- ***p*值**

  在线性模型的参数估计中，通常用p值评估一个参数的估计值是否显著。对于某个参数β，通常设定一个原假设：$\beta = 0$，β估计值的p值就代表了在原假设成立的条件下，出现观察样本（通常以估计值的t值表示）或者更极端样本的概率。因此，如果p值越小则代表观察样本支持原假设的力度越小。也就是说，p值越小（比如说小于0.05）我们就可以以相当大的概率（95%）认定原假设并不成立，β显著不为0。

- **AIC**（赤池信息量准则）

  赤池信息量准则的定义公式为：

  $$\mathrm{AIC} = 2k - 2\ln(L)$$

  其中k是模型中的参数个数，$\ln(L)$是最大似然函数值。AIC越小模型的效果越好。

- **BIC**（贝叶斯信息量准则）

  贝叶斯信息量准则的定义公式为：

  $$\mathrm{BIC} = k \cdot \ln(n) - 2\ln(L)$$

  其中k为模型中的参数个数，n是样本数量（在“追龙”的例子中，就是用户数量），$\ln(L)$是最大似然函数值。与AIC一样，BIC越小代表模型的效果越好。

- 熵

  关于熵，我们将在7.4.4节中详细介绍。

实际操作

因为分步回归本身的特点，选出的最佳变量子集很容易过拟合数据：模型的样本内拟合效果很好，但是样本外的预测效果却不然。

在应用变量选择标准时，并不需要在每一步都重新训练一遍模型，这会相当费事。由于每个选择标准都可以写成函数的形式，根据函数的泰勒展开式，可以从理论上表示出选择标准如何随着变量个数的改变而改变。

最后，我们还是想提一下，领域专家应该在变量选择过程中起一定的作用，在用算法选择变量之前，不妨咨询一下他们的看法。

7.4.4　决策树与嵌入型变量选择

决策树方法的吸引力在于它非常直观。除数据科学领域以外，人们在日常生活中做决定时，也可以将大问题分解成一系列小问题，逐个解决。比如下图7-3，一位大学生在决定自己的时间分配问题时所用到的决策树。

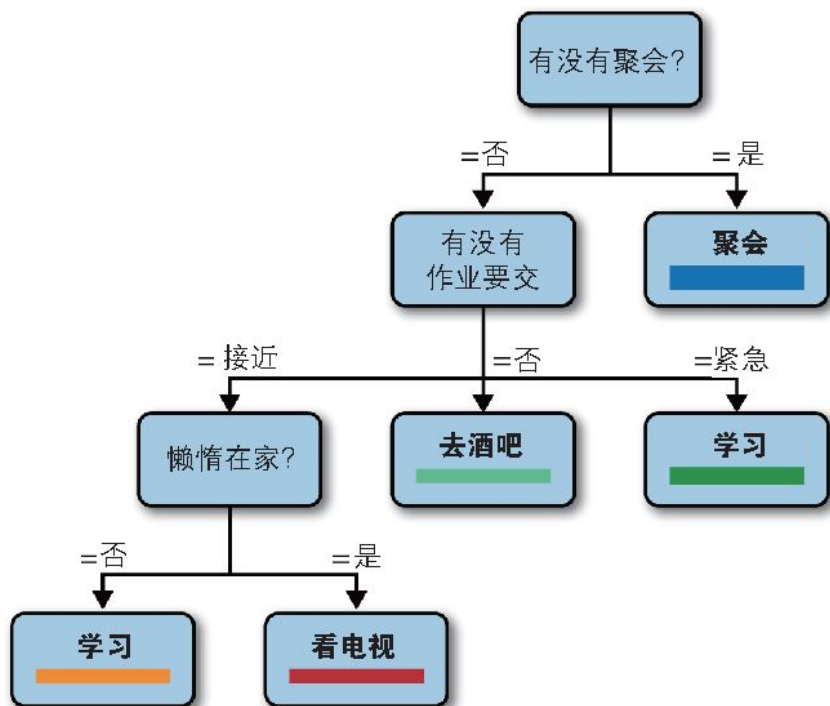


图7-3：一个大学生在解决自己的时间分配问题时用到的决策树（原图摘自Stephen Marsland的著作*Machine Learning: An Algorithmic Perspective*（《基于算

法的机器学习》，Chapman and Hall/CRC），并获得了作者的许可

该决策树的整体结构取决于好几个元素：今天有没有派对、今天有没有作业要交、今天的心情如何，等等。可以看出，决策树的结构清晰、导向明确、非常容易理解，因此具有十分优良的可解释性。这也是决策树如此受欢迎的主要原因。

在数据科学中，决策树一直用来处理分类问题。以之前“追龙”的用户分析问题为例，我们想预测用户下一个月是否会续订。这是典型的分类问题，预测变量只取两个值：“1”代表用户会续订，“0”代表用户不会续订。要想预测用户下一个月续订行为，需要考虑很多因素（比如用户杀死龙的数量、用户的年龄性别、用户在游戏上花去的时间等）。决策树告诉我们可以把这些因素对预测变量的效用用如图7-3所示的结构图表示出来，最终得到类似图7-4的决策树。

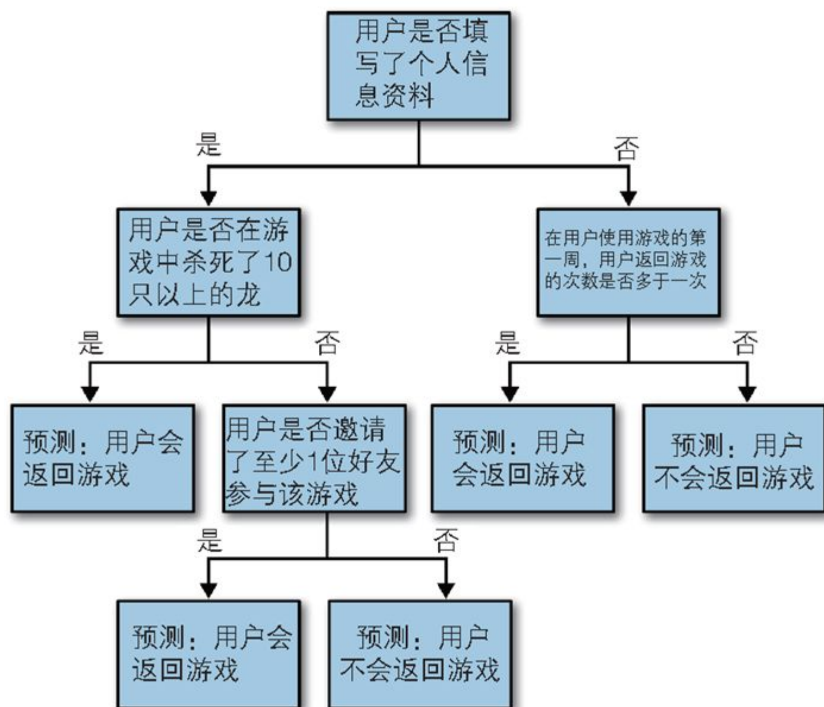


图7-4：“追龙”问题的决策树图

那么一个自然的问题就是：如何把这些变量合理地放在决策树上呢？哪些先放、哪些后放、什么时候停止摆放，等等。的确，建立决策树

有诸多细节需要考虑。变量的摆放和拆分要基于数据中的信息，而不是拍脑袋瞎猜。总体原则是，在每一步，都先放上“信息量最大”的变量，从上往下地搭建决策树。因此，一个关键性的问题便是，如何认定一个变量的“信息量”大小呢？

为了说明这个问题，我们用X表示数据集中的特征变量，并假设其只能拆分为两类：0和1。$p(X = 1)$以及$p(X = 0)$分别代表两类的概率值。

7.4.5 熵

在信息论中，熵用来定量表示一个特征变量所包含的信息量，其定义如下：

$$H(x) = -p(X = 1)\log_2(p(X = 1)) - p(X = 0)\log_2(p(X = 0))$$

$H(X)$代表变量X的熵值。注意，当$p(X = 1)$或者$p(X = 0)$时，X的熵为0。这同下面的极限性质一致：

$$\lim_{t\to 0} t \cdot \log(t) = 0$$

因此，只要X取两类中任何一类的概率为0，则X的熵值为0。此外，由于$p(X = 1) = 1 - p(X = 0)$，因此X的熵关于0.5对称，并在$X = 0.5$处取得最大值1。图7-5是X的熵与$p(X = 1)$的函数图，$H(X)$的对称性和极值点一目了然：

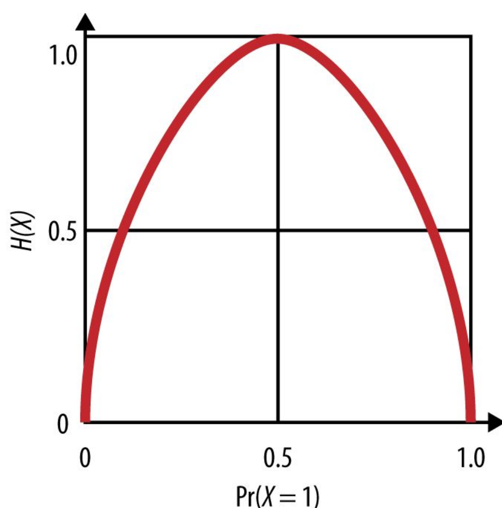


图7-5：变量X的熵值图

从数学上定义的熵值很难被直观地理解，就连“熵”这个词本身就很容易让人抓狂。它到底指的是什么？我们说过，熵代表一个变量所包含的信息量的大小，那上面关于熵的定义公式与信息量大小到底有何关系？

信息量大小与变量的不确定性有着直接关系，一个变量的不确定性越高，其熵值越大10。这里不妨举个简单的例子。假设X表示一个出生婴儿的性别，$X = 1$表示男孩，$X = 0$表示女孩。从概率论的角度来说，生男生女的概率是等同的，因此$P(X = 1) = P(X = 0) = 0.5$。这对于我们来说，基本上没有任何信息含量，也就是$X$具有高度的不确定性，因此其具有大的熵值。这与图7-5的结论不谋而合，当$P(X = 1) = 0.5$时，X的熵达到最大值1。

> 10在信息论中，这称作系统的混乱程度。

相反，如果用X表示沙漠中某天下雨的概率，从常识上来讲，既然是沙漠，那么下雨的几率肯定非常小，甚至接近于0。对于这样的变量X，由于其不确定性很低，因此熵值也很低。从图7-5来看，其熵值接近于0。

既然熵可以代表一个变量所包含的信息量，我们便可以以它为目标函数，最优化模型的参数。比方说，用X表示用户是否续订我们的产品，那么我们必然想知道什么样的变量对X的影响最大。这里我们需要定义信息增益（Information Gain, 用IG（X, a）表示）的概念，对于变量X，在给定一个变量值a的情况下，X熵值的减少量：

$$IG(X,a) = H(x) - H(X|a)$$

$H(X|a)$即代表X给定属性a的条件熵。假设a_0是属性a的实际属性值，那么$H(X|a = a_0)$表示在该属性值给定条件下X的条件熵：

$$H(X|a = a_0) = -p(X = 1|a = a_0)\log_2(p(X = 1|a = a_0)) - p(X = 0|a = a_0)\log_2(p(X = 0|a = a_0))$$

假设属性a有n个属性值，那么$H(X|a)$的计算公式为：

$$H(X|a) = \sum_{a_i} p(a = a_i) \cdot H(X|a = a_i), i = 1, 2, \ldots, n$$

用通俗的话讲，X给定属性a的条件熵指的是在知道属性a的取值之

后，X熵值的减少量。因为熵代表了不确定性，也就是说，在知道a之后，我们对变量X有了新的认识，对它的不确定性减少了。从属性a的角度来说，就是它为我们更进一步了解X所带来的额外信息量。

有了熵和信息增益的概念之后，我们就可以顺利地搭建决策树了：在摆放变量时，总是优先摆放信息增益量最大的变量，因为它能够为我们了解X带来最大的信息量。

7.4.6 决策树算法

决策树是一种迭代算法，先从第一个根节点开始选择变量进行拆分，直到所有的变量都已用尽，或者在某节点上只能对等拆分[10]时，停止迭代。在每一个节点处，都选择可以最大化信息增益的变量用于拆分。

> [10]拆分后的两类包含的数据量相等。

一个完整迭代的决策树往往会过拟合，为了解决这个问题，可以采用“事后剪枝法”。也就是说，为了防止决策树过于复杂，在构建好完整的树之后，在树的某个节点处将树剪短。剪短的决策树往往具有更好的外推扩展能力。

决策树中隐藏了一个变量选择的过程，我们称为“嵌入型变量选择”。这也意味着，决策树模型不需要使用传统的“过滤型”或者“包装型”的方法选择变量。因为，当模型使用信息增量选择变量进行拆分时，已经自动地选择了对模型本身而言最为重要的变量。因此，变量的选择过程已经巧妙地嵌入了决策树模型的搭建过程。

回到刚才“追龙”的例子当中，我们已经搜集了大量的用户数据，并且生成了许多特征变量。我们关心的目标变量是用户在下一个月是否会选择续订该游戏。rpart是R中做决策树模型的标准软件包之一。利用其中的rpart函数，我们可以迅速搭建一个决策树模型，并画出相应的决策树图。里面的代码可供参考：

```
# 利用rpart函数建立分类树模型
library(rpart)

# 运行分类树模型
model1 <- rpart(Return ~ profile + num_dragons +
num_friends_invited + gender + age +
num_days, method="class", data=chasingdragons)
```

```
printcp(model1) # 展示模型结果
plotcp(model1) # 交叉验证结果可视化
summary(model1) # 转换为二元变量时阈值选择的详细结果

# 分类树可视化图
plot(model1, uniform=TRUE,
main="Classification Tree for Chasing Dragons")
text(model1, use.n=TRUE, all=TRUE, cex=.8)
```

7.4.7 如何在决策树模型中处理连续性变量

12软件包在搭建决策树模型的时候已经为我们考虑到了连续性变量的情形，并自动进行了最优离散化的操作。如果没有软件的帮助，你需要弄明白如何最优地离散化连续性变量，并为决策树所用。

> 12决策树中的每个节点都会被拆分，因此对于离散型变量来说，拆分操作可以进行得非常自然：只需要按照变量的类别拆分就可以。如果需要在决策树中使用连续性变量来说，其关键就在于如何将变量离散化，选择若干阈值将变量拆分到一些子区间中，并以区间为类。

离散化关键在于阈值的选取。比如变量*X*指的是用户在游戏中杀死龙的个数，若选取10作为阈值，则*X*可以分为一个包含两类的二元变量（1代表杀死龙的个数大于10，0代表小于10）。被离散化之后的变量可以按照刚才讨论的方式摆放到决策树中，这不是问题。阈值的选取当然不能随意化，这会对模型的效果有着直接影响。

因为有众多的可能性，比方说，离散化为二元类还是多元类，以多大的间隔挑选阈值等。阈值的选取是一个相当棘手的问题。对它的选取，本身可以视作决策树模型的一个子模型。至于到底如何选取，这样取决于数据和问题本身的特点。

泰坦尼克号乘客的生存模型

关于决策树模型，Will推荐我们看一看BigML网站上有关泰坦尼克号乘客生存问题的决策树模型（见图7-6，参考http://goo.gl/YsyWJW）。原始数据和源代码都来自于Encyclopedia Titanica（http://www.encyclopedia-titanica.org/）。图7-6只是该决策树模型的一小部分，Encyclopedia Titanica提供了更为详细的交互化决策树图，感兴趣的读者不妨去他们的网站看一看。

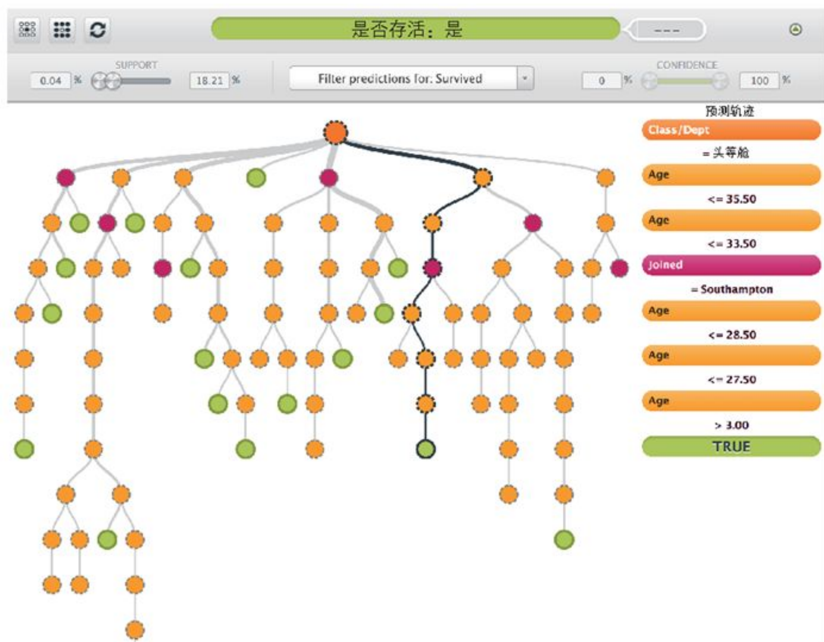


图7-6 泰坦尼克号乘客生存模型的决策树图

7.4.8 随机森林

随机森林是决策树模型的拓展，它基于**bagging**（袋装）模型。bagging的全称是**bootstrap aggregating**（解靴集成法）。该方法可以显著地提高模型的精度和稳健性，但会牺牲决策树模型本身的最大优点：可解释性。人们经常批评随机森林的黑匣子特征，认为从模型形式上来说，其基本不可以解释。然而，从模型设置来看，随机森林模型非常简单，它只有两个参数：N代表森林中决策树的个数；F代表每棵树上使用的（随机选取的）特征变量个数。

在深入了解随机森林模型之前，我们有必要先了解一下bootstrapping（解靴法）。解靴是一种可放回的抽样方式，一个解靴样本（bootstrap sample）就是从原数据中有放回取处的n个数据，因为抽样是可放回的，因此一个数据可能会被抽取多次，一些数据很可能一次也不会被抽到。至于n的大小，一般取所有数据量的80%，当然这没有严格的规定。对于随机森林来说，n可以理解为第三个参数。

建立随机森林模型，一般遵循以下两个步骤。

1. 从原始数据中抽取N个不同的解靴样本，每个样本都建立一个决策树。每个决策树只随机使用F个特征变量。

2. 每个决策树的搭建都遵循之前讨论的原则，包括变量的选择，连续性变量的处理等。

每一个决策树都可以进行“事后剪枝”以避免过拟合，但对于随机森林模型来说，这完全没有必要。随机森林的一大特性就是对过拟合的免疫性：模型本身就吸收了数据中的特质方差，因此即使不做单个决策树的“事后剪枝”，随机森林模型也不会过拟合。

下面提供一段随机森林模型的R代码：

```
# 作者：Jared Lander
#
# 这里使用的数据是来自ggplot2中的diamonds数据集
require(ggplot2)

# 载入和大致观察一下diamonds数据
data(diamonds)
head(diamonds)

# 画出直方图，在直方图横轴$12 000的位置加画一条竖直线
ggplot(diamonds) + geom_histogram(aes(x=price)) +
geom_vline(xintercept=12000)

# 构造一个包含TRUE/FALSE的二元变量
# 用来表示该变量的取值是否超过一个阈值
diamonds$Expensive <- ifelse(diamonds$price >= 12000, 1, 0)
head(diamonds)

# 删除price列
diamonds$price <- NULL

## 此处需要加载glmnet包
require(glmnet)
# 创建预测变量矩阵
# 其中最后一列变量被剔除，因为它是模型的因变量
x <- model.matrix(~., diamonds[, -ncol(diamonds)])
# 构建响应向量
y <- as.matrix(diamonds$Expensive)
# 运行 glmnet
system.time(modGlmnet <- glmnet(x=x, y=y, family="binomial"))
# 画出模型的参数图
plot(modGlmnet, label=TRUE)

# 这里设定一个随机数种子值
# 目的是为了结果的可重复性
# 感兴趣的读者可以多尝试运行几次该程序，得到的结果应该是一样的
set.seed(48872)
```

```
sample(1:10)

## 决策树模型
require(rpart)
# 构建一个简单的决策树
modTree <- rpart(Expensive ~ ., data=diamonds)
# 决策树模型分叉可视化
plot(modTree)
text(modTree)

## bagging模型（全称是bootstrap aggregating）
require(boot)
mean(diamonds$carat)
sd(diamonds$carat)
# 下面的函数对均值进行bootstrapping操作
boot.mean <- function(x, i)
{
        mean(x[i])
}
# 这样我们便可以计算均值估计的方差了
boot(data=diamonds$carat, statistic=boot.mean, R=120)
require(adabag)
modBag<-bagging(formula=Species~.,iris,mfinal=10)

## boosting模型
require(mboost)
system.time(modglmBoost <- glmboost(as.factor(Expensive) ~ .,
            data=diamonds, family=Binomial(link="logit")))
summary(modglmBoost)
?blackboost

## 随机森林模型
require(randomForest)
system.time(modForest <- randomForest(Species ~ ., data=iris,
importance=TRUE, proximity=TRUE))
```

特征选择有必要吗？

对于特征选择，有些人觉得没有必要，他们常说：与其费力气做特征选择，还不如花点时间多捞点数据。这也不无道理，比如在用过滤型的方法做变量选择时，如果以简单相关系数为标准，很可能会选出一些与目标因变量高度相关的变量，但他们本身可能毫无关系。在预测标准普尔指数的收益率时，人们发现孟加拉国的黄油产量与其有很强的相关性，但这明显是伪相关。类似的例子还有很多，这些都是由于相关系数本身的限制所造成的。然而，如果有更多的数据，类似这样的伪相关关系可能就没那么重要了。

但是从另外一个角度来看，变量选择其实至关重要。统计模型都会有“偏差-方差折中效应”，也就是说越简单的模型可能方差很小，但是其模型偏差很大，而越复杂的模型可能越精确，也就是其偏差可能很小，但是模型方差很大，扩展能力很弱，容易引起模型过拟合。一个好的模型其实就是在偏差与方差中寻求一个最佳平衡点。因此，过多的数据，过多的变量并不会带给我们更好的模型。变量选择对于建模来说，是十分必要的。

7.4.9　用户黏性：模型的预测能力与可解释性

众所周知，决策树模型有着优良的可解释性。也就是说，模型建立之后（要保证其预测能力尚可），可以通过决策树图的形式解析模型。从模型中，可以发现自变量与因变量之间的逻辑联系。

但是，从决策树图中我们可能会得到这样的逻辑联系：用户在第一个月玩该游戏的时间越长，其在下一个月续订的可能性越大。如果费尽心思建立的模型只能得到这样常识化的解释，分析人员肯定会非常失望。即使不用模型，我们也知道一个人玩游戏的时间越长，代表他越喜欢这款游戏，其续订的可能性当然更大。但是如果模型能够告诉你，在游戏开始的5分钟插播广告会减少客户续订的概率，但是如果在用户玩了一个小时游戏之后再插播广告则不会显著地影响客户续订的概率。这时候模型才会显得比较有魅力。因为你从其中得到了你原本不知道的信息，这条信息明确地指示你，在游戏刚开始的5分钟切莫插播广告。这样的模型解释性才是真正有用的。当然，为了证实用户是否确实不喜欢在游戏刚开始的5分钟内看到插播的广告，还需要进行A/B测试（见第11章）。但是最起码，模型指出了可能改善用户体验的方向，这才是我们建模的目的所在。

模型的解释可能充满陷阱。这里我们有必要把特征变量分成两类，一类是由用户的行为产生的行为变量（比如用户一个月玩了10次该游戏），另一类是游戏开发人员的行为变量（比如每天插播广告10次，或者龙的颜色从绿色改成了红色等）。在解释模型的时候，要注意这两种类型变量之间可能存在的相关/因果关系及其对模型解释力的影响。比如说，模型的结果显示用户在游戏中得分的高低与其续订游戏的可能性大小有很强的相关关系。那么为了招揽客户，吸引他下个月继续续订，我们是否可以通过给用户增加游戏得分的方式获取更多的用户续订呢？显然不能！用户在游戏中得到的分数与他的续订行为只存在相关关系，而并非因果关系。其中有一个潜在的干扰变量悄然地

联系着用户的得分高低与他的续订行为。这个干扰变量就是用户对游戏的喜爱程度，或者上瘾程度。因为用户喜欢这个游戏，玩上瘾了，他才会取得高分，他才会愿意在下个月开始的时候继续续订这个游戏。因此，即便通过变量选择我们可以更好的解释模型，但是真正与改善用户体验相关的，是在考虑到用户的行为变量的影响之后，如何改变开发人员的行为（比如，插播更少的广告，而不是提高用户的游戏得分）。

7.5 David Huffaker：谷歌社会学研究的新方法

David的工作重点是有效地把定量与定性研究、大数据与小数据研究结合起来，充分发挥各自的优点。大数据研究应该以小数据为起点，先在小数据上建立对问题的基本认知和把握，再延伸到对大数据的整体研究上。反之亦然，在大数据上发现的数据特征，也应该及时地回溯到小数据上进一步验证审查，在一小部分样本人群中做“可用性研究”。此外，也可以用小数据里发现的新东西为大数据特征润色。在小数据上进行的探索性分析也应该联系到大数据上，并与现有的学术文献成果结合，在大数据中找到对应点。这种定量与定性、大数据与小数据的遥相呼应，应该成为数据科学研究的典范模式。

Rachelz在谷歌时曾与David共事，他们当时的合作非常成功。由于他们知识构成相互补足，当他们在Goolge＋项目（谷歌的社交应用）与一些优秀的软件工程师和计算机科学家一起合作时，总是会碰撞出创意的火花，并取得了很大的成功。David作为一个社会科学家，为团队带来了社会学分析的视角。对于在线社交行为的定量分析与理解，他也同样非常在行。他博士毕业于美国西北大学，研究方向有关媒体、科技与社会。在我们的课堂上，David与同学们分享了谷歌的社交研究团队的工作方式，他着重提出，一个优秀数据科学团队应该把定量与定性研究、大数据与小数据研究相结合才能取得成功。

谷歌的工作方式非常开放，不同背景的人组成团队，攻克同一个项目，这种混搭的方式会产生巨大的生产力。学术研究与项目开发的界限在谷歌已经被极大程度地弱化，很多项目都带有强烈的学术研究色彩。谷歌团队在2012年6月就他们的研究和工作方式甚至发表了一篇名为“Google's Hybrid Approach to Research”（“混搭研究在谷歌”，参见http://goo.gl/ejtPw2）的论文。在谷歌的产品开发队伍中，研究团队是重要的组成部分。谷歌的产品总是在不断实验、不断完善的。工程师们从产品立项的第一天起就尽量保证代码的质量，产品在小范

围内实验后，会慢慢地开放给大众用户。因为谷歌的产品总是拥趸众多，因此他们不可能总是从头再来，产品开发的每一步都要脚踏实地。在小范围客户中验证产品可行性之后，再慢慢地扩大用户群，并在此过程中不断获取用户反馈并及时改善产品，这就是谷歌的产品开发哲学。

7.5.1 从描述性统计到预测模型

David指出，作为数据科学家，我们的工作决不能只停留在描述性统计阶段，应该勇于跨入深水区，设计实验，研究变量之间深层次的关系。这需要我们敢于把工作重心从描述性统计转移到预测模型上来。

他举了一个在谷歌内部发生的例子。Google+的一个重要功能叫作“圈子”，在设计之初，谷歌的研究人员已经知道人们在分享照片或者新闻时有着强烈地选择性。比如，人们更倾向于把照片分享给亲人，一些八卦只会与自己的好朋友分享。这就是谷歌设计“圈子”的初衷。然而，即便这个功能听起来似乎能够满足人们的需求，但是谁也不能肯定用户一定会喜欢上这个功能。人们分享的动机和方式因人而异，用户的心理其实很难揣摩。

为了验证产品的可行性，谷歌采用了一个混合研究方式，使用了形形色色的研究方法，包括小型的定性研究，还有大规模的定量研究，最后确认了产品的可行性和应该具备的主打功能。

谷歌其实主要采取了问卷调查和访谈的研究模式，但细节上更为复杂。为了确定什么样的“圈”名最流行，谷歌随机挑选了10万名用户，从其中挑选出168位最为活跃的用户，请他们填写一份调查问卷，并对其中的12位用户进行了细致访谈。为了消除取样时的选择偏差，他们对访谈的深度相应地也做了调整。

他们发现，大多数人都会选择性地分享，大多数人都会使用“圈子”的功能，而“圈子”的命名方式大多跟工作或者学校的名字有关，有时候从“圈子”的名字就可以看出这帮人是群死党还是群刚认识的朋友（其中一个圈子的名字叫“一辈子老顽童组”，很明显是一帮死党老朋友，另一个圈子的名字叫作“新东方同学”，很明显是一帮刚从英语课上认识的学生）。

在问卷中，他们询问了用户为什么会分享内容。回答基本可以分为三类：其一是因为想展现自我，包括自己的经历、观点等；其二是用户具有参与某个对话的欲望；其三是一些天生就乐于分享的人。

他们还询问了用户在选择分享对象时，都有哪些考虑。回答也可以分为三类：其一是私密性：有些信息只会分享给最亲的好友；其二是相关性：一般信息只会分享给与信息相关的人，或者对信息感兴趣的人；最后是覆盖度：有些人分享只为了尽可能大的覆盖所有的好友，甚至公众，以扩大自己的影响。

总而言之，用户的分享行为确实具有相当的选择性，主要的影响因素是分享的内容和分享的对象。针对这个特点，谷歌在打造“圈子”的产品主打功能时，应该着重于优化用户控制分享的内容以及对象上的体验。只要做好了这些用户最关心的功能，最终的产品才会取得成功。

思维实验：大规模社交网络分析

在第10章中，我们会和John Kelly一起详细学习社交网络分析的内容，现在让我们稍微热身一下。我们刚刚已经看到了Google＋团队在设计产品时，针对用户的选择性分享行为开展了卓有成效的用户可行性研究。想一想，如何把该研究的结论应用到大数据层次上？现在较为流行的做法是使用网络图，对于Google＋来说，每一个用户代表一个节点，“圈子”内的用户之间可以用有向连接线相连。在小范围的调查结束之后，你需要想一下如何将调查的结论拓展到大数据上：需要记录哪些数据、构造什么假设并设法在网络图中找到答案。

作为一个数据科学家，应该拓展思维空间，用不同的结构和形态表示数据，这样你会发现，即使是网络图，也可以画出各种不同的模样，代表数据中不同的特征。

下面是网络的几种不同形态。

- 每一个节点都代表网络世界中（Second Life，参见http://secondlife.com/）的用户，节点之间的连接线代表用户之间的互动。因为用户之间的互动可以有许多不同的形式，因此也应该允许节点之间的连接线有不同的类型。
- 每一个节点代表一个网站，连接线代表网站之间的超链接。
- 节点代表一个定理，而连接线代表定理之间的关系（参见这个例子：http://

www.math.columbia.edu/~dejong/plaatje.png）。

7.5.2 谷歌的社交研究

谷歌的所有产品都带有强烈地“社交”元素，甚至连谷歌的搜索引擎也是如此：比如说，你搜索某个条目，谷歌会告诉你，你的某个朋友也对这个条目点了赞，这称作社会化标注。通常人们更关心专家们，而不是亲戚朋友朋友所标注的信息。这很简单，假设你准备买一瓶酒，你肯定更想看到品酒专家对某款酒的标注，而不是你的某位亲友。

但是如果问，在品酒专家和亲友之间你更加信任谁，你肯定会选择亲友。其实也就是说，“点赞”的朋友并不一定是亲密的朋友，人们在社交网络上所表现出来的行为和情绪有着强烈的复杂性。到底用什么来度量社会化标注的重要程度，还是要取决于标注本身的特点。

通常，数据科学家喜欢用诸如“点击率”这样的指标衡量其重要程度。

7.5.3 隐私保护

社交网络上最为棘手的问题就是用户的隐私保护问题。谷歌曾经做过一项调查，问题是关于人们对社交网络隐私保护的看法：比如隐私顾虑是否会削弱他们参与社交网络的热情；通常的隐私顾虑都包括哪些内容，等等。

调查结果显示，用户对隐私的顾虑会直接影响他们参与社交网络的热情。这一点都不奇怪，因为社交网络的急速传播性，用户会顾虑他们分享的每一条信息，关心什么时候分享这条信息比较合适，信息还会被哪些好友分享，等等。在社交网络上，我们对信息毫无控制权，当你面对这么多烦恼的时候，你很容易产生负面情绪，对参与社交网络的兴趣大减。

下面列出调查中用户列出的一些顾虑。

- 身份信息窃取
    - 银行卡信息被窃等会导致潜在的金钱损失
- 数字世界
    - 个人信息

    - 个人搜索历史
    - 垃圾邮件
    - 私人大尺度照片（被老板看见就惨了）
    - 不情之请
    - 垃圾广告

- 真实世界

    - 离线骚扰
    - 威胁家人
    - 被人跟踪
    - 失业风险

7.5.4 思维实验：如何消除用户的顾虑

用户的以上顾虑其实细细看来都比较合理，Rachel课上的同学对这些顾虑展开了头脑风暴，并给出了以下一些解决方案。

- 在产品说明页详细地写明相关的隐私政策协议。谷歌确实这么做了，但好像基本上没有人愿意花时间读这些协议。
- 可以把推荐的内容更加人性化地推送给用户，比如说："尊敬的用户，基于你对某某条目点了赞，我们猜您可能也对这条信息感兴趣。"但是用户对此仍会心存疑虑。
- 可以告诉用户，所有用户数据的有效期为一年。在一年之后所有数据都会从服务器上永久删除。

其实，只要把数据政策透明地呈现给用户，用户的很多疑虑都会消除。但关键问题是，怎么让这些政策更加透明化呢？下面是学生们讨论的几点建议。

- 将数据的使用状态用动态图形或者动画的形式展示给客户。
- 用户需对自己的隐私有控制权。
- 用户可以快速设置相关的隐私选项。
- 设置中往往会存在很多不人性化的"霸王条款"，可以考虑把这些条款做得更加人性化一点。
- 最为理想化的情形是，所有的默认设置都充分考虑到了用户的隐私顾虑，因此用户不需要作任何改动。

David在最后留了一句话给大家，作为本章的结束：当你向大数据迈进的路上，不要被定量分析蒙蔽了双眼。简单的、基于逻辑的定性分析对于更加深刻地理解数据同样重要。像定性调查分析这样的传统工

具，有时候反而能取得奇效。

第 8 章　构建面向大量用户的推荐引擎

推荐引擎，又名推荐系统，是数据科学的典范应用之一。日常生活中，人们或多或少都同推荐系统打过交道，比如在亚马逊买书时，系统会推荐你可能感兴趣的图书，Netflix会推荐你可能喜欢的电影。因此推荐引擎是向门外汉介绍数据科学及其应用的一个很好的切入点。然而，很少有人会思考隐藏在推荐系统背后的算法细节，更不会有人意识到当他们在网上买书或者评价电影时，相应的数据都自动生成并提供给了推荐系统，这些数据反过来又会提高推荐系统的准确性，为人们推荐更多更好的内容。

之所以称推荐系统为“典范”的数据科学应用，不仅是因为它是一个典型的数据推动的应用，还在于构建该系统时需要一些数据科学的两项必备技能：线性代数和编程。同时，推荐系统还展示了一些看似直观的解决方案在面对大数据时可能会面临的计算量上的挑战。

本章，Matt Catts将和我们一起分享他在为Hunch.com构建推荐系统时所积累的经验。在构建这种直接面向用户的大型推荐系统时，他是基于何种考量做出某些决定的；在面对各种算法的时候他又是如何做出取舍的，我们都将一一揭晓。

Matt毕业于麻省理工学院，专业是计算机科学。毕业后供职于SiteAdvisor公司，随后与人合伙创建了Hunch公司（http://hunch.com/），自己担任CTO。Hunch是一个为用户作各种推荐的网站。首先，系统会问用户一系列问题（从Hunch网站的数据来看，人们似乎很乐意回答网站提出的问题），根据用户对这些问题的回答，Hunch会为每一位用户构建一个推荐系统。如果用户反过来问系统一个问题，比如“我该买一个什么样的手机？”“这次我该到哪儿去旅游？”，系统就会给出为该用户量身定制的、合理的建议。利用机器学习技术，随着系统的不断学习，这些推荐的质量会变得越来越好。Matt的工作，就是负责整个推荐引擎背后的技术开发和研究（R&D）。

起初，他们会把问题设计得尽量有趣以争取更高的回答率。后来发现，这些问题的设计应该以尽可能获取最多的用户信息为宗旨。经过不断地调整，他们发现只需要问用户不超过20个问题，推荐系统的预

测准确率就能达到80%。有些问题很常见，而有些问题对于用户来说可能会比较出乎意料，比如会问你的性格是争强好胜还是随遇而安、内向还是外向、理性或感性，有点像MBTI的人格测试。

后来，Hunch决定不直接向用户提问，转而向互联网抓取数据，并以API形式向外提供服务。服务可以被第三方用来为既定网站提供个性化的推荐内容。无疑，这是一个更好的商业模式，最终Hunch成功出售给了eBay。

Matt从小就开始写程序，这是他的强项，除了推荐系统，他并非所有领域的专家，而Hunch作为一个数据公司，需要跨领域、多学科知识的人才。

因此，Matt说过一句很经典的话："一个人不可能完成所有工作，组建数据团队就像召集十一罗汉一样，个个都得身怀绝技。"[1]

> 1《十一罗汉》是一部有名的美国电影。

8.1　一个真实的推荐引擎

推荐系统的适用范围十分广泛。电影公司会根据用户的观影历史，推荐给客户他们可能会喜欢的电影；在线购书网站根据客户过往的购书记录为用户推荐他们可能喜欢的新书；旅游公司会根据用户以前旅游过的地方，为他们建议一次新的旅程。这样的例子数不胜数。

构建这样的系统虽然方法很多，但总体感觉起来其实都大同小异。本章会为大家介绍如何实现这样一个系统，我们的实现虽然简单，但是麻雀虽小，五脏俱全。

假设有一个用户的集合U，待推荐的项目定义为集合V，如Kyle Teague在第6章中指出的那样，可以用二分图（见图8-1）表示两个集合之间的关系，如果用户评价过某件东西，就用一条线将二者连接起来，这种评价有好有坏，评价可以是正面的，可以是反面的；可以是连续的，可以是分散的，我们可以给线加上权重以表示这些连接在某种意义上的"强度"值。评价系统可能会很被设计非常复杂，但这里我们不讨论过于复杂的系统设计。

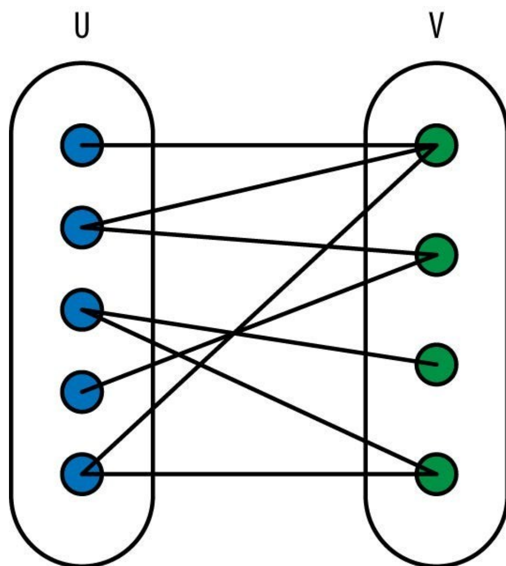


图8-1：推荐系统的二分图：左侧是用户，右侧是推荐的项目比如电视节目

推荐系统所需要的训练数据集其实就包含在上面的二分图中：主要说来，就是用户的特征变量以及用户对某些商品的偏好数据。有了这些数据之后，基于推荐系统的分析，就可以为用户推荐新的商品了。最终具体的推荐项目就是推荐系统的模型输出结果。

你可能会获取的数据包括一些用户或商品的元数据，比如用户性别，商品的颜色。当用户登录访问你的网站时，需要注册账号，你就可以得到用户的性别、年龄和个人嗜好等数据，还可以让他们选择自己喜欢的三件商品。

用户可以表示为一组特征向量，该向量有时包含用户所有的特征变量（包括用户的元数据以及用户的其他特征变量），有时可能只包含用户的元数据或者只包含用户的喜好数据，这取决于你想用这组特征向量干什么。如果向量只包含用户的喜好数据，那么得到的向量可能会十分稀疏，因为不同用户的喜好会差别很大。在得到每位用户的特征向量之后，可以把他们绑在一起，形成一个大的用户向量矩阵，我们不妨称为U。（之前的用户集合也叫作$\mathbf{U}$，这里我们有点滥用数学符号的嫌疑。）

8.1.1　最近邻算法回顾

第3中我们讨论了最近邻算法，这里让我们就推荐系统问题对这个算

法稍作回顾：如果你想预测用户A是否喜欢某商品，你可以问问A的“邻居”B的意见。如果B喜欢该商品，那么很可能A也会喜欢该商品。换句话说，用户A的相似用户是用户B，用户A没有看到过或者买过某种商品，而用户B买过，那么我们就可以向用户A推荐该商品，因为基于最近邻原理，他的邻居买过此商品，那么他很可能也会对此商品感兴趣。

第3章我们说过，最近邻算法严重地依赖于“距离”的定义。对于用户的喜好变量（是一个二元变量）可以用Jaccard距离来测度用户之间喜好的相似程度（即“1 - (二人都喜欢的商品个数／只有其中一人的商品个数)”）。另外，余弦度和欧式距离也是合理的选择。



到底哪种距离测度最好？

这个问题我们已经提过很多次了，答案是：没有严格、统一的标准。具体问题需要具体对待，我们建议在实际操作中尝试不同的测度，再通过模型评价选定一个较为合理和有效的。

8.1.2　最近邻模型的已知问题

最近邻模型固然是一个不错的模型，它足够简单，模型逻辑也很容易理解：为了给某用户推荐某商品，最近邻模型会先找该用户的“邻居”，并根据这些“邻居”的喜好情况为该用户设计推荐内容。但是有很多已知的问题限制着该模型的适用范围，我们这里列举一些。

- 维度诅咒
  当数据的维度过高时，即便是最近的“邻居”在高维空间中也会离得很远。

- 过拟合问题
  如果用$k = 1$的最近邻算法会很容易产生过拟合的问题。因为离你最近的那个“邻居”所能提供的信息十分有限，它本身可能包含大量“噪声”信息。日常操作中，我们很少使用$k = 1$，$k = 5$是一个较为常见的选择。但是过大的k同样也会带来更多“噪声”信息。

- 多重相关问题

数据中的特征变量很多，他们中间很多是高度相关的。多重相关问题会带来信息重叠问题，比如年龄大的人往往倾向于比较保守的性格，如果把用户的年龄和保守程度都用在模型中，就会造成某项信息的过度使用。信息的重叠和过度使用都会带来欠佳的模型表现。如果能把数据中的特征变量精简，只采用一些重要的、信息没有重叠的（不多重相关的）的变量，对于提升模型效果是大有裨益的。

- 特征变量之间的相对重要性
  某些变量可能比另外一些变量更加重要，这是很容易理解的。比如你的年龄对于预测你对商品1可能没有其他变量重要，但是在预测你对商品2的时候可能又至关重要。根据研究对象的不同，要给不同的变量赋予不同的权重，这对于提高模型效果也通常是有帮助的。至于如何确定权重，可以看看变量之间的协方差矩阵。

- 稀疏性
  如果特征变量的向量（或者特征变量矩阵）过于稀疏，或者有过多的缺失值，那么也就意味着数据中的绝大多数信息是未知的。如果用Jaccard距离这样的测度就很难得到有用的信息。

- 测量误差
  数据的测量总有误差（也称作报告误差）：因为人们是会说谎的。

- 计算的复杂度
  计算是有成本的——成本取决于计算问题本身的复杂度。[2]

- 距离测度指标的敏感性
  有些距离指标对于变量的取值范围是很敏感的。比如欧氏距离就有这个问题：年龄变量之间的欧氏距离就要远远大于二元变量之间的欧氏距离。也就是说，由于欧氏距离对变量取值的敏感性，它的应用受到了较大的限制。

- 时变效应
  用户的喜好可能会随着时间的改变而改变，这样模型就不再适用了。比如在eBay，如果一个用户刚买了一个打印机，那么在短时间之内他可能只想买墨盒。过了一段时间他又会想买别的商品。

- 模型更新成本

随着新数据的加入，我们需要不断地更新模型。而模型更新的成本往往比较高。

> 2近邻模型在数据量较大时的计算成本是很高的。

上面所列举的诸多问题中，前两项是最为重要的：维度诅咒和过拟合问题。我们应该如何对付这两个问题呢？我们接下来以线性回归模型为出发点，探讨一下这个问题。

8.1.3 超越近邻模型：基于机器学习的分类模型

我们接下来把机器学习模型简化，针对每一个商品使用单独的线性回归模型用于预测。在某个模型中，我们的目的是给定某位用户的属性变量，预测该用户是否会喜欢该商品。比如，某个模型被单独用来预测你是否喜欢电影*Mad Men*（《广告狂人》），另外一个模型被单独用来预测你是否喜欢歌手Bob Dylan（鲍勃·迪伦）。

用$f_{i,j}$表示用户*i*对商品*j*的喜好情况（或者*j*代表用户*i*的元数据，比如年龄和是否是注册用户等，而i则是用户的其他属性变量）。这个表示方法可能会让人有点摸不着头脑，所以你要花时间稍微消化一下，我们在这里把元数据也当作了某种意义上的“商品”。我们之前也有过这样的表述，如果你实在觉得很难理解，也没有关系，继续往下看。当我们说推荐系统可以预测你的喜好时，从模型层面上来说这里的“喜好”是广义的，可能是你的某种属性变量。比如说，也许你在注册时没有填写性别或者我们没有问你的性别是什么，那么“性别”可以作为一个待预测变量，也就是这里的“喜好”。

为了让这种表述变的更容易理解，我们假设每个用户都有三种属性变量，也就是说对于任何用户i都有$f_{i,1}$，$f_{i,2}$和$f_{i,3}$。为了预测该用户i对于某个新商品的喜好（暂时用p_i表示该喜好），应用以下的线性回归模型：

$$p_i = \beta_1 f_{1,i} + \beta_2 f_{2,i} + \beta_3 f_{3,i} + \epsilon$$

好消息是，对于这样一个多元线性回归模型的参数估计，我们已经学习过了。参数的估计可以使用线性代数的方法或者最优化的方法。这是线性回归模型的统计推断，我们在之前已经遇到过了很多次。

但是坏消息是，这样的模型是针对单个商品的，也就是说，有多少商品就要有多少个类似的线性回归模型。而且，由于模型都是独立的，因此这样的模型没有考虑进商品之间可能存在的相关关系，这对于数

据分析本身来说，就意味着大量的有关商品之间关联的信息被忽略和浪费了。

等等，我们之前说过的变量的权重问题在线性回归模型中得到了很好得解决，因为回归模型的参数就是天然的权重估计值。

但是更严重的坏消息是，这样的模型很容易产生过拟合的问题。从表现形式上来说，如果对某个商品的喜好的数据量不足，那么在该商品的回归模型中，参数的估计值会变得很大。



我们刚才说到如果模型中的某些参数的估计值过大，那么就认为中模型出现了过拟合，这里做一下解释。试想在模型中我们使用了两个一模一样的变量，其中一个变量的参数估计值可能为100 000而另外一个是-100 000，从参数估计的绝对值来看，似乎两个变量都十分重要，但我们知道其实其中一个变量是完全多余的。因此在建模之前，我们最好做一些关于参数估计大小上下限的先验假设，一切超过该上下限的参数估计值都是可疑的，很可能意味着模型已经出现了过拟合。

为了解决过拟合问题，可通过设定贝叶斯先验信息的方式强制所有的参数估计值都落在一定范围之内。从模型形式上来看，我们可以惩罚过大的参数估计值。惩罚大参数等同于在数据的协方差矩阵上加了一个先验信息矩阵（参见http://mathbabe.org/2013/02/24/the-overburdened-prior/）。最后的模型解只取决于一个参数，通常用λ表示。

接下来一个自然的问题就是，该怎么选择这个惩罚参数λ呢？第3章的模型交叉验证的方法也可以用在这里：选取一部分数据作为训练数据，一部分用作模型验证与评价。不停地变换λ的值，直到模型的评价标准告诉你已经找到了一个较为合理的值。现实中我们经常使用这样的交叉验证方法，但是却很难保证一定能找到那个最优的λ值。



有些变量因为取值本身就比较大，如果用惩罚的方法，对它们可能非常不公平。解决这个问题最简单和直接的办法就是在建模之前把所有的变量正则化，我们在第6章讲过正则化的概念。这样所有的变量都在同一个水平线上取值，惩罚就变得相对公平了。正则化的操作可以根据变量意义的不同而有所差异，至于到底在正则化中使用什么均值和方差，建模者可以自由选择。从贝叶斯统计的角度来说，正则化操作等同于在模型中添加了某种形式的“先验信息”。最后一个关于先验信息的问题是：虽然将先验信息添加到模型中，只要λ够大，模型一般都会有唯一的最优解。但是过大的λ可能会导致模型中绝大多数甚至全部的参数估计都接近于0，这样模型本身的实际意义就荡然无存了。

8.1.4 高维度问题

我们已经解决了过拟合的问题，第二个棘手的问题就是数据的高维度问题了：比如刚才的数据中可能包含上百万商品的信息，这样的维度会给模型带来很多问题。解决高维度问题的两大杀手锏是奇异值分解（Singular Value Decomposition）和主成分分析（Principal Component Analysis），接下来我们会详细讲解。

在讲解其中的理论细节之前，想一想在现实生活中我们是如何通过“隐含变量”（latent feature）的形式达到降维目的的。我们经常说某人很“酷”，而这个概念是很难量化和直接观测到的，我们称为“隐含变量”。而“酷”本身可能是很多因素构成的，比如这个人的行为方式，语音语调等，这些因素往往是可以直接观测到的。因此一个“酷”就涵盖了许许多多可观测变量的信息，这从形式上来说，就达到了降维的效果。

在这个降维过程中发生了两件值得注意的事情：其一是很多变量被一个隐含变量所涵盖，其二是这个隐含变量不可观测。

从数学上来讲，对于包含很多可观测变量的数据矩阵来说，我们并不知道其中的隐含变量意味着什么、该如何解释等，我们甚至不知道其

中有几个“重要”的隐含变量。所有的一切都是由算法和计算机自动完成的。“重要”在这里指的是该隐含变量可以解释数据中绝大部分的方差。如果可以用很少的“隐含变量”就能解释数据中绝大部分的方差，那么这些隐含变量就是“重要”的。

从推荐系统的角度来看，降维的目的是找到用户的“品味”（taste information）。根据每个用户品味的不同，系统可以做出个性化的推荐。因此，“品味”在这里是一个隐含变量，并且提取自关于用户的可观测的变量。

不过，因为很多属性变量都来自于系统在用户注册时的问卷调查数据，也就意味着很多变量是简单的二元变量。而二元变量所包含的信息往往少于连续性变量和多类别变量。Hunch的研究人员就发现，问卷中多设计一些比对问题（comparison question）能够带来更好的推荐效果。

> 是时候好好学习一下线性代数了！
>
> 如果不太买线性代数的账，或者对于其中诸如“秩”（rank）、“正交”（orthogonal）、“转置”（transpose）、“基”（base)、“张空间”（span）和“矩阵分解”（matrix decomposition）这些概念的几何解释都非常陌生，那么本章之后的内容你一定会感觉一头雾水。
>
> 从空间和矩阵的观点看数据，我们能够更深刻地洞悉数据。理解诸如矩阵的变换和子空间等概念对深入理解某些模型的数学细节，或者对于优化某些模型的源代码都大有帮助。矩阵角度的模型洞察力甚至决定了一个数据初创公司的技术实力，这也是Hunch能被eBay收购的主要原因：他们拥有厚实和前沿的技术实力。如果你觉得自己的线性代数学的不够好，我们推荐大家去听一听Khan Academy（可汗学院）上的线性代数课。

8.1.5　奇异值分解（SVD）

我们已经热身很久了，下面就让我们直接进入奇异值分解的数学细节。给定一个秩为k，维度为$m \times n$的矩阵X，根据线性代数的理论，矩阵X可以分解成三个矩阵的积：

$$X = USV^T$$

其中U、S和V的维度分别为$m \times k$、 $k \times k$和$k \times n$。U和V矩阵内的列是相互正交的，S是一个对角矩阵。SVD在线性代数中的标准描述是把U和V称作为方酉矩阵（square unitary matrix），而把S称作方阵（rectangular matrix）。我们也沿用这个叫法。之后我们将通过减秩的方法尽可能的近似矩阵X，同时达到降维的目的。关于SVD分解存在性的证明可以在奇异值分解的维基百科页面上找到（http://goo.gl/GLS6sG）。

现在让我们把SVD分解与之前推荐系统的数据分析问题联系起来。X对应之前的原始数据集，里面包含用户和商品的信息，其中有m位用户和n件不同的商品，X的秩为k。k也是X中可能包含的隐含变量个数d的上限值。d是SVD需要确定的调整参数（tuning parameter）。

SVD分解中的矩阵U的每一行代表用户，矩阵V的每一行代表商品。S矩阵的对角线上的元素叫作“奇异值”（singular value），他们的大小衡量的是相应位置上隐含变量的重要性：最重要的隐含标量对应最大的奇异值。

8.1.6 关于SVD的重要特性

因为矩阵$\boldsymbol{U}$和$\boldsymbol{V}$各自的列向量之间是相互正交的，因此可以根据奇异值从大到小的顺序将他们变换位置。因为奇异值的大小代表着相应位置上隐含向量的重要性，因此在排序之后可以考虑扔掉奇异值很小的那部分矩阵而只保留奇异值较大的那部分，它们代表了原数据矩阵的大部分信息。

也就是说，我们把$\boldsymbol{S}$矩阵的对角线元素按照从大到小的顺序重新排列了之后扔掉了其右下方的大部分元素，同理$\boldsymbol{U}$和$\boldsymbol{V}$也相应地做了重新排序和舍弃。如果d是一个远小于k的数，那就代表我们舍弃了$\boldsymbol{U}$、$\boldsymbol{S}$和$\boldsymbol{V}$的绝大部分元素，只保留了对应大奇异值的那一小部分元素。计算机图像学中经常讲到的“压缩”（compression）概念就是如此。原矩阵$\boldsymbol{X}$的绝大部分信息被保留了下来，但是矩阵的规模却缩小了很多。矩阵$\boldsymbol{S}$的对角线元素具有明确的含义，矩阵$\boldsymbol{U}$和$\boldsymbol{V}$中的值也同样有很好的解释。

这样的矩阵分解对于推荐系统来说到底有何用处呢？设想矩阵$\boldsymbol{X}$中已经填满了用户对商品的打分数据。（通常来说，矩阵中最好不要出现打分为0或者没有打分的情况，没有打分就意味着缺失值，而SVD不能处理有缺失值的矩阵）应用SVD分解完矩阵之后，我们想要的并不

是三个独立的分解矩阵，我们的目的其实是为了预测新的用户对某商品的喜好。而这个预测可以用矩阵$\hat{X}$来表示，而它恰好就是我们刚才讨论过的X的近似矩阵。

在8.1.2节中我们讲到了最近邻模型的很多已知的缺陷，比如缺失值问题和计算量大的问题。SVD也有这两个方面的问题。SVD涉及的矩阵分解对计算机的计算能力要求颇高，因此对于大型矩阵SVD会显得力不从心。我们应该想一想如何改进SVD的计算速度问题。

8.1.7 主成分分析（PCA）

保留SVD中的$\boldsymbol{U}$矩阵和$\boldsymbol{V}$矩阵，如果我们可以抛弃$\boldsymbol{S}$矩阵而只通过这两个矩阵近似原矩阵$\boldsymbol{X}$，则有：

$$X \equiv U \cdot V^T$$

如果这个近似是可行的，那么我们总是希望$\boldsymbol{X}$与$U \cdot V^T$之间的误差尽可能小。这是一个最优化问题，我们希望最小化他们之间的误差平方和：

$$argmin \sum_{i,j} (x_{i,j} - u_i \cdot v_j)^2$$

此处，u_i表示$\boldsymbol{U}$矩阵对应用户i的某行，v_j对应$\boldsymbol{V}$矩阵中商品j的某行。同样，$\boldsymbol{V}$矩阵的行上可以包含用户的元数据变量，比如年龄变量。

那么点积$u_i \cdot v_j$就代表用户i对商品j喜好程度的估计值，我们当然是希望其与真实值越接近越好。

也就是说，你想找到一组最优的$\boldsymbol{U}$和$\boldsymbol{V}$矩阵，以最小化预测值与实际观测值之间的平方误差。实际观测值代表你已经观测到的，对其有充分了解的值，那么这里的逻辑就是，如果它们对于已经观测的值有良好的表现，那么对于没有观测到的值，其表现也会十分不错。你应该已经对这个逻辑相当熟悉了——这就是均方误差的概念，我们之前在线性回归里已经仔细地阐述过了。

这里唯一需要确定的参数就是d，它的数值代表了隐含变量的个数。矩阵U的行代表用户，列代表隐含变量；而矩阵$\boldsymbol{V}$的行代表商品，列代表隐含变量。

到底如何选择d呢？在我们的例子中，一个合适的d大概在100左右。

我们之前说过在用户注册填写问卷调查的时候，20个问题就足够了解该用户了，因此100似乎看起来有点过大了。d的大小可以人为设定，只要不出现过重的计算负担，我们建议将d设在100左右。



得到的隐含变是原矩阵在n维空间上的基向量。但是一般来说，如果原矩阵中包含过多的缺失值会导致隐含变量没有唯一解。但是这没有关系，因为我们只是想得到其中的某一个解而已。[3]

> 3从矩阵理论上来说，隐含变量解的形式与原数据矩阵中的缺失值并无关系，因此译者不能理解原书作者这段话到底想表达什么意思。

定理：隐含变量是互不相关的

在k近邻那一章我们讲过变量之间的多重相关问题，在选择进入模型的变量时，没有建模者希望在模型中包含进任何冗余变量。隐含变量的好处就在于它们是互不相关的。下面我们简单的证明一下：

假设我们已经找到了矩阵$\boldsymbol{U}$和$\boldsymbol{V}$，满足$U \cdot V = X$具有最小化的平方误差。这样的矩阵$\boldsymbol{U}$和$\boldsymbol{V}$可能有很多，我们想要的是其矩阵元素和最小的矩阵$\boldsymbol{U}$和$\boldsymbol{V}$。因此我们可以尝试最小化矩阵$\boldsymbol{U}$和$\boldsymbol{V}$内元素的平方和。不难看出，我们可以通过右乘矩阵$\boldsymbol{U}$一个可逆矩阵$\boldsymbol{G}$（$d \times d$），相应的左乘矩阵$\boldsymbol{V}$一个矩阵G^{-1}也可以得到相同的矩阵$\boldsymbol{X}$，因为：$U \cdot V = (U \cdot G) \cdot (G^{-1} \cdot V) = X$

假设矩阵$\boldsymbol{G}$的行列式值为1，也就是说我们在对矩阵$\boldsymbol{U}$做变换的时候强制了其是体积不变的（volumn-preserving transformation）。如果我们暂时忽视矩阵$\boldsymbol{V}$中元素的值，而只关心最小化矩阵$\boldsymbol{U}$中元素值的大小，那么就相当于在一个n维空间内最小化一个d面体的表面积（该d面体的体积是固定的）。实现的唯一方法就是让该d面体的每一条边都相互正交，也就是说*U矩阵内的每一列都是互不相关的。

但是不要忘记了，我们刚才似乎忽略了矩阵$\boldsymbol{V}$！但是基于同样的道理，如果我们把注意力集中在矩阵$\boldsymbol{V}$上，而选择忽略矩阵$\boldsymbol{U}$，那么$\boldsymbol{V}$的每一行必须是互不相关的。其实SVD从理论上已经告诉了我们$\boldsymbol{U}$矩阵列之间以及$\boldsymbol{V}$矩阵的行之间是相互正交的，但其实这里的$U \cdot V$与X的SVD还是有所不同。有些人觉得$U \cdot V$就代表着$\boldsymbol{X}$的奇异值分解，其实

从严格意义上来说是不对的。

现在放松矩阵的$\boldsymbol{G}$的条件，允许它的行列式的值为任意值——比如说，我们允许$\boldsymbol{G}$是单位矩阵的倍数形式。那么只需要一点微积分的技巧，就可以发现最佳的倍数值（最佳指的是可以最小化矩阵$\boldsymbol{U}$和$\boldsymbol{V}$的元素平方和）是矩阵$\boldsymbol{U}$和$\boldsymbol{V}$所有元素的几何平均值。

这就是我们的证明过程，你被说服了吗？

8.1.8　交替最小二乘法

那么到底如何找到这样的矩阵$\boldsymbol{U}$和矩阵$\boldsymbol{V}$呢。从上面的叙述来看，我们似乎是先通过最小化误差平方和，再通过最小化矩阵$\boldsymbol{U}$和$\boldsymbol{V}$的元素平方和，才能找到最佳的矩阵$\boldsymbol{U}$和$\boldsymbol{V}$。从步骤来看，是两个一先一后的步骤。但其实，这两步可以同时完成。

基本上来说，这是一个最优化的问题，但是这个最优化问题不同于普通最小二乘问题，它是没有解析解的。我们需要类似于“梯度下降法”这样的迭代算法去解决这个最优化问题。只要这个最优化的对象函数是凸函数，那么就基本可以保证算法可以最终收敛到真值。（也就意味着，算法不会在局部最优点上永久停留，而找不到全局最优点。）即便函数本身不是一个凸函数，我们也可以通过增加惩罚项的方式把函数强制地变为一个凸函数。

下面就是这个迭代算法的详细步骤。

- 随机生成一个矩阵$\boldsymbol{V}$。
- 将矩阵$\boldsymbol{V}$固定，最优化矩阵$\boldsymbol{V}$。
- 将矩阵$\boldsymbol{U}$固定，最优化矩阵$\boldsymbol{U}$。
- 重复上述步骤知道矩阵$\boldsymbol{U}$和矩阵$\boldsymbol{V}$中元素的变动不再显著为止。具体来说，我们可以定义一个容忍值ϵ，只要变动幅度不超过该容忍值，我们就认为该迭代算法已经“收敛”了。

没有证明过程的定理：如果先验信息量足够，那么刚才的迭代算法一定收敛

先验信息量越大，最优化过程会变得越简单。但是从另一方面来说，如果先验信息太多，那么所有的参数估计都会接近于0，这对于实际问题来说没有任何意义。因此，我们从来不希望先验信息变得过于主导。但是，我们不可以选择一个最优化的先验信息量，因为如果我们这么做，先验信息就不能称为先验信息了。我们在最小化参数的过程

中还要想着如何找到一个能够近似矩阵$\boldsymbol{X}$的近似矩阵，这两个过程其实是南辕北辙的。你对参数的大小关注得越多，对矩阵$\boldsymbol{X}$的近似就要关注得少一点。但是，我们还是认为，更多的精力应该放在如何更好地近似矩阵$\boldsymbol{X}$上，这才是我们的真正目的。

8.1.9 固定矩阵$\boldsymbol{V}$，更新矩阵$\boldsymbol{U}$

在固定矩阵$\boldsymbol{V}$，最优化矩阵$\boldsymbol{U}$的那一步，最优化的过程是关于用户的。对于每一个用户i，最优的对象是：

$$argmin_{ui}\sum_{j\in P_i}(p_{i,j}-u_i*v_j)$$

其中v_j是固定的。换句话说，这里只关心用户i。

这看起来与最小二乘法的数学表达没有本质上的区别！的确是。根据最小二乘法的结论，上面最优化过程的最优解为：

$$u_i=(V_{*,i}^TV*,i)^{-1}V_{*,i}^TP_{*,i},$$

其中$V_{*,i}$是矩阵V的子集，其代表用户i对商品的喜好程度。上式中的求逆操作并不难，因为矩阵$V_{*,i}^TV*,i$是一个d阶方阵，是一个较小的矩阵。因为每个用户的商品喜好向量并不是一个很长的向量，因此这一步的计算量其实很小。

刚才讲到的迭代过程是固定$\boldsymbol{V}$更新$\boldsymbol{U}$的过程。类似地，如果固定矩阵$\boldsymbol{U}$更新矩阵$\boldsymbol{V}$，其结论相同。求逆操作只涉及一个d阶方阵。

另外一个好消息是，因为用户之间的喜好是相互独立的，所以上述迭代过程完全可并行化。如果计算机的CPU是多核的，或者手头有几台空闲的计算机，你完全可以用并行化的方法提高算法迭代的速度。这对于大数据分析来说至关重要。

8.1.10 关于这些算法的一点思考

每一种方法都有不同的实现版本，我们已经展示了如何在一些问题上做出权衡以达到更好的预测效果。有时候，针对不同的问题也要做相应的调整，凡事还是要具体问题具体对待。

比如说，随着新用户和新数据的加入，我们可能要不断地更新矩阵U

和矩阵V，不断地最优化这两个矩阵。但是对于某些用户，经过细心地观察，你可能会决定他们的状态已经相对比较稳定，因此不需要再更新关于他们的那部分模型以节省计算资源。这些决定都要取决于你自己。

就像所有的机器学习模型一样，交叉验证的方法应该贯穿于建模的始终——我们要始终抽离出一部分数据用作模型的独立验证。这是有效避免过拟合问题的关键方法。

8.2　思维实验：如何过滤模型中的泡沫

对用户喜好的预测是通过最小化误差的形式实现的，这对大家有何启示呢？推荐系统的模型，从表现形式上来看会对模型的更新产生什么样的影响呢？

具体来说，商品展现给用户的先后顺序是否会影响商品本身的受欢迎程度呢？或者进一步说，商品本身是否会因为这样的人为操作而具有本身不该具有的泡沫优势呢？如果有的话，我们又如何在模型中过滤掉这些泡沫效应呢？

大家不妨静下心来好好想一想这些问题。

8.3　练习：搭建自己的推荐系统

在第6章中，我们有幸拿到了GetGlue公司提供的数据，练习了有关探索性数据分析的内容。现在，我们不妨再次利用这个数据集，搭建一个属于你自己的推荐系统。下面的Python代码并不是针对该数据集的，而是Matt为大家提供的一段示例代码。读懂这段代码，并尝试把它应用到GetGlue数据集上。

Python示例代码

```
import math,numpy
pu = [[(0,0,1),(0,1,22),(0,2,1),(0,3,1),(0,5,0)],[(1,0,1),
(1,1,32),(1,2,0),(1,3,0),(1,4,1),(1,5,0)],[(2,0,0),(2,1,18),
(2,2,1),(2,3,1),(2,4,0),(2,5,1)],[(3,0,1),(3,1,40),(3,2,1),
(3,3,0),(3,4,0),(3,5,1)],[(4,0,0),(4,1,40),(4,2,0),(4,4,1),
(4,5,0)],[(5,0,0),(5,1,25),(5,2,1),(5,3,1),(5,4,1)]]
pv = [[(0,0,1),(0,1,1),(0,2,0),(0,3,1),(0,4,0),(0,5,0)],
[(1,0,22),(1,1,32),(1,2,18),(1,3,40),(1,4,40),(1,5,25)],
[(2,0,1),(2,1,0),(2,2,1),(2,3,1),(2,4,0),(2,5,1)],[(3,0,1),
(3,1,0),(3,2,1),(3,3,0),(3,5,1)],[(4,1,1),(4,2,0),(4,3,0),
```

```
(4,4,1),(4,5,1)],[(5,0,0),(5,1,0),(5,2,1),(5,3,1),(5,4,0)]]

V = numpy.mat([[ 0.15968384, 0.9441198 , 0.83651085],
[ 0.73573009, 0.24906915, 0.85338239],
[ 0.25605814, 0.6990532 , 0.50900407],
[ 0.2405843 , 0.31848888, 0.60233653],
[ 0.24237479, 0.15293281, 0.22240255],
[ 0.03943766, 0.19287528, 0.95094265]])

print

U = numpy.mat(numpy.zeros([6,3]))
L = 0.03

for iter in xrange(5):

    print "\n----- ITER %s -----"%(iter+1)
    print "U"
    urs = []
    for uset in pu:
        vo = []
        pvo = []
        for i,j,p in uset:
            vor = []
            for k in xrange(3):
                vor.append(V[j,k])
            vo.append(vor)
            pvo.append(p)
        vo = numpy.mat(vo)
        ur = numpy.linalg.inv(vo.T*vo +
            L*numpy.mat(numpy.eye(3))) *
            vo.T * numpy.mat(pvo).T
        urs.append(ur.T)
    U = numpy.vstack(urs)
    print U

    print "V"
    vrs = []
    for vset in pv:
        uo = []
        puo = []
        for j,i,p in vset:
            uor = []
            for k in xrange(3):
                uor.append(U[i,k])
            uo.append(uor)
            puo.append(p)
    uo = numpy.mat(uo)
        vr = numpy.linalg.inv(uo.T*uo + L*numpy.mat(num
            py.eye(3))) * uo.T * numpy.mat(puo).T
    vrs.append(vr.T)
V = numpy.vstack(vrs)
print V
```

```
err = 0.
n = 0.
for uset in pu:
    for i,j,p in uset:
        err += (p - (U[i]*V[j].T)[0,0])**2
        n += 1
    print math.sqrt(err/n)
print
print U*V.T
```

第 9 章　数据可视化与欺诈侦测

本章内容的贡献者是Mark Hansen和Ian Wong。Mark是哥伦比亚大学的教授，Ian之前（2012年11月我们课程进行的时候）是Square公司的统计学家，现在任职于Prismatic公司。他们两位虽然都住在加利福利亚，并且十分注重数据的可视化，但在本章他们主讲的题目将截然不同。同其他章的贡献者一样，他们两位都是深刻的思想家，他们常常思考的问题包括：什么是好的代码、编程语言作为一种人类的表达方式到底有什么样的特性。当然，也包括本书最重要的主题：到底什么是数据科学？

9.1　数据可视化的历史

Mark Hensen来自加利福利亚大学洛杉矶分校（UCLA），在休假期间，他兼职于《纽约时报》研发实验室，并且是哥伦比亚大学布朗媒介创新研究院的院长。他博士毕业于伯克利大学统计学院，毕业之后在贝尔实验室工作了几年，随后去了UCLA，任职于统计学部，并且同时任职于电子工程部和设计/媒体艺术部。Mark在数据可视化界有相当高的知名度，同时他也是一个富有激情的演说家。某个晚上，他本来应该是在星巴克喝大杯拿铁的，却被我们拽到课堂中给大家上了一课。

Mark会首先带我们回顾一下数据可视化的历史，最后会跟我们分享一些他的数据可视化项目。他可视化项目的成果现在都实实在在地摆在了一些博物馆或者广场上供人欣赏。他的工作哲学和理念深深地影响着他身边的人。他也在教授一些关于数据可视化的课程，他的工作往往突破常规、不拘一格，并且深深地影响着这个行业。可以说，他是数据可视化的先驱，这也是Rachel把他请到课堂中来的原因。在本章的最后，我们还会讲一些关于数据可视化技术前沿的内容。

9.1.1　Gabriel Tarde

Mark首先提到了社会学家Gabriel Tarde。Tarde觉得社会科学产生的数据量要远远大于别的学科。他指出，其他学科对数据的态度往往倾向于隔靴搔痒，各种模型的存在也无非是对数据某种形式的变换和汇总：比如说，生物学家并不直接研究细胞，而是间接地研究细胞表现出来的某些功能。Trade觉得这一定程度上是对信息匮乏的妥协。从

生物学家的角度来看，我们真正应该研究的应该是每一个细胞本身产生的所有数据。

在社会学中也一样，每个人就是社会学中的细胞。在Facebook上，每个人每天都在生产大量的数据，这是研究人和社会的绝佳媒介。

但是这里自然会有一个问题：当我们太注重细节和个体的时候，很可能会失去对事物整体的把握。在社会学研究中，如果太注重个体研究，我们很容易忽略人们之间的文化属性和社交属性，而这些属性往往意义更加深刻。法国现代社会学家Bruno Latour在他一篇评论Tarde的文章“Tarde's Idea of Quantification”（Tarde的量化心得）中这么说过：

> 整体，作为一个概念，可以看作其局部成分的某种方式的重构或者转化。同样的局部构件，在不同的重构和转化工具下会表现出不同形式的整体。

——Bruno Latour

Tarde于1903年在某个类似日报的媒体上发表的一段话，可以说是对Facebook的一段精准的预言。他说：

> 如果统计学能够按照近几年的发展速度发展下去，如果信息的到来能够更加准确、迅捷、海量和有规律，那么一个信息化的时代终将到来。在这个时代里，社会上每一个小事件完成后都会自动生成数据并上传到统计储存器中。这些数据将会实时、连续地通过媒体向公众传递，甚至会通过更为形象化的方式向海外传播。到了那个时候，我们每瞄一眼报纸或者海报，都会被巨量信息所包围。无论是社会发展的细微动态、商业竞争的即时信息、政党支持率的偏移，甚至是某种学说的兴衰，都将以统计数据来说明问题，人们对社会的认识会变得更加精简浓缩。未来的我们，对信息的获取和处理或将成为我们本能的一部分，就像我们现在睁开眼睛就可以看到人与自然一样。在未来，信息或将成为自然的一部分。

——Tarde

Mark于是在他的课堂中用Brune的一句话如是总结道：

> 改变了研究工具，就等于改变了整个社会学研究的整体面貌。

——Bruno Latour

Mark说（Tarde应该也这么认为），这就类似薛定谔的猫，我们应该重新审视我们观察和了解这个社会的方式，以及我们认识局部与整体关系的角度，因为当我们观察这个世界的时候，它其实是在不停变化的，我们需要有动态的、全局的视角。

换句话说，老式的数据收集方式迫使我们使用样本和汇总统计量（比如均值）去间接地了解我们所观察的对象。但是现在呢？我们得到的数据越来越多，有时候甚至可以掌握总体的全部数据，我们自然应该改变我们的研究工具，而不应该墨守成规，总是想着用抽样和汇总的方法，我们的研究触角既可以深入到个体，也可以扩大到整体。比如我们将要在下一章中讲到的基于图论的社交网络研究，它会带给我们更多的关于研究个体本身以及研究个体之间关系的信息。这些都源于信息潮的到来，以及我们处理信息能力的增强。既然旧有的限制条件已经慢慢消失了，我们自然应该改变我们思考和研究问题的方式。

9.1.2　Mark的思维实验

随着数据的个人化特征愈发明显，更多关于人类活动本身的数据需要我们去分析。由此产生的问题就是，我们应该使用什么样的工具研究人与人、人与社会、社会与国家、国家与世界之间纷繁复杂的关系呢？

民意调查和总统支持率这样的数据分析调查框架固然可以掌握公众意见的动向，但能否展示民意中个性化和互动的部分？

即便可以，我们又是否愿意生活在这样一种环境中——我们的个性和互动信息能够被精准记录下来并随时供人取用？

9.2　到底什么是数据科学

Rachel在第1章中提出并尝试回答了“到底什么是数据科学”这一问题。Mark在这里想从一个全新的角度再一次审视这个命题。他首先提到了John Tukey的一句名言：

> 作为一个统计学家的最大特权就是，你可以在每一个

人家的后院玩耍。

——John Tukey

如果把统计学家，抑或数据科学家，比作可以在任何人的后院玩耍的全能型人才，这样合理吗？换句话说，这样的“全能”真的是成为数据科学家的必要条件吗？到底什么样才能算作“全能”呢？

我们不妨把学科分为“数据类传统学科”和“其他学科”两类，其中前者主要包括数学、统计学、计算机科学等。“全能型”从概念上来说不过是代表对于所有的数据类传统学科的知识都谙熟于胸，因为我们不同于常人并深深地痴迷于数据科学的各个传统领域。如此看来，我们或许是过于自大了，从而导致我们看待这个问题的角度也变得非常狭隘。

因此即使是对于最简单的问题，即到底哪些学科是“传统”的数据类学科，哪些属于“其他”类，我们都还回答不上来。

在Mark看来，“其他”类应该包括大部分社会科学学科和理科类的学科，比如教育学、设计学、新闻学和媒体艺术学等。我们要成为的，不仅仅是单纯的“技术专家”，因为每一项技术本身其实都来自于某个研究领域对工具的实际需求。比如说，地理信息系统是由地理学家的需求衍生和发展出来的，而文本数据挖掘的技术则来自于数字人文学科的实际需求。

也就是说，每个领域的实际需求在引领着不同技术的发展。数学这样的从理论出发的学科固然重要，技术也对各个学科的发展起着至关重要的作用。当学科与数据科学交叉融合之时，都会衍生出具有学科特色的需求，进而不断地推动学科本身以及数据科学不断向前发展。

数据科学的健康发展离不开每个学科领域独立的健康发展，学科交叉和融合也变得愈发普遍和不可阻挡。在这样时代性的交叉和融合进程中，数据科学应该担当起旗手的重任，其要面对的一大难题就是什么样的交叉和融合才是健康的。

因此，Mark在描述他自己的数据科学背景的时候指出，数据科学家是一个地地道道的“扩张主义者”，对这个概念你也许不会感到诧异。

9.2.1 Processing

Mark在给艺术家和设计师的编程课中提到过一种叫作

Processing（http://processing.org/）的编程语言。他以Processing为例，展示了设计师和工程师在编程时的思维区别。一门好的数据编程语言应该以分析人员的思维方式为起点，以分析任务为导向，并且具有完整的结构性。

我们可以通过一个思维实验来进一步探讨这个问题：艺术家或者设计师到底需要什么样的编程语言？R在统计学家和数据科学家中流行，是因为R解决了他们对于某些分析任务的刚性需求，比如随机变量、分布类、向量和数据结构等。相比于R，一门艺术家或者设计师使用的编程语言应该解决哪些刚性需求呢？

艺术家更加关心形状、轮廓等人的大脑中可以想象的图像元素，给艺术家的编程语言应该能够最大程度上以最直观的形式把人脑内的图像元素外化并且展现出来。素描、3D、动画甚至是交互的方式都可以，最重要的是能够清晰和直观地展示给人看。

Processing就是这样一门基于Java的编程语言，它是专门给艺术家设计的。Mark说他在给艺术家和设计师讲编程课的时候，要从最基本的编程技巧说起，包括迭代、`if`语句等。因为这些基本的编程技巧在我们看来可能再平常不过，但是对于艺术家和设计师来说却是一种崭新的知识形态。因此，在给这些人上课的时候，他需要回到原点，从零开始。

9.2.2 Franco Moretti

Mark接下来讲到了近距离和远距离文本阅读的问题。他首先提到了斯坦福大学的文学家Fanco Moretti。

Franco认为“远距离阅读”的效果就是不需要逐行逐句逐字地读，而只需要远远地看一下就能理解文本的大致含义。这种阅读方式有点类似于对文本内容的降维（第8章中我们学习了一些典型的降维技术）。

Mark觉得Franco的例子很好地说明了数据科学如何在不同的学科发挥功效。与其越俎代庖，不如以一个旁观者的姿态细心学习各个学科的特点，并结合数据科学的要求将二者很好地结合起来。这样一种学习和旁观的态度对数据科学的健康发展起着方向性的作用。

9.3 一个数据可视化的方案实例

下面给大家展示一些被Mark所称道的可视化案例。对于每一个案例，

我们都应该问这样一个问题：可视化的数据是什么？这个方案是该问题最理想的可视化方案吗？

图9-1是一个关于城市能源使用量的可视化项目。图中烟囱中投射出来的绿色阴影的大小代表了该城市中能源的使用量。



图9-1：Helen Evans和Heiko Hanse的Nuage Vert可视化案例（http://youtu.be/l_4rTQCWItw）

在另外一个叫作One Tree的项目中（见图9-2），设计师Natalie把一个种子克隆了许多份并栽培在城市的不同区域。种子在不同的环境下成长，几年后就变成了一棵棵大树。这个案例形象地展示了环境对于植物生长的长期影响。







图9-2：Natalie Jeremijenko的One Tree可视化案例（http://boingboing.net/2003/05/16/natalie-jeremijenkos.html）

图9-3的案例叫作Dusty Relief，图中的几何状建筑物可以收集其周围的污染物并以粉尘的形式表现出来。



Dusty Relief, New Territories

图9-3：New Territories的Dusty Relief项目（http://www.new-territories.com/roche2002bis.htm）

《纽约时报》的研发实验室开发了一个叫作Project Reveal的可视化项目（见图9-4）。 图中的镜子内置了一个智能的面部识别系统，只要站在镜子对面，镜子上就会显示关于你的一些个人信息。Mark说每当站在这个镜子前面的时候，那种感觉是非常奇幻的，难以名状。

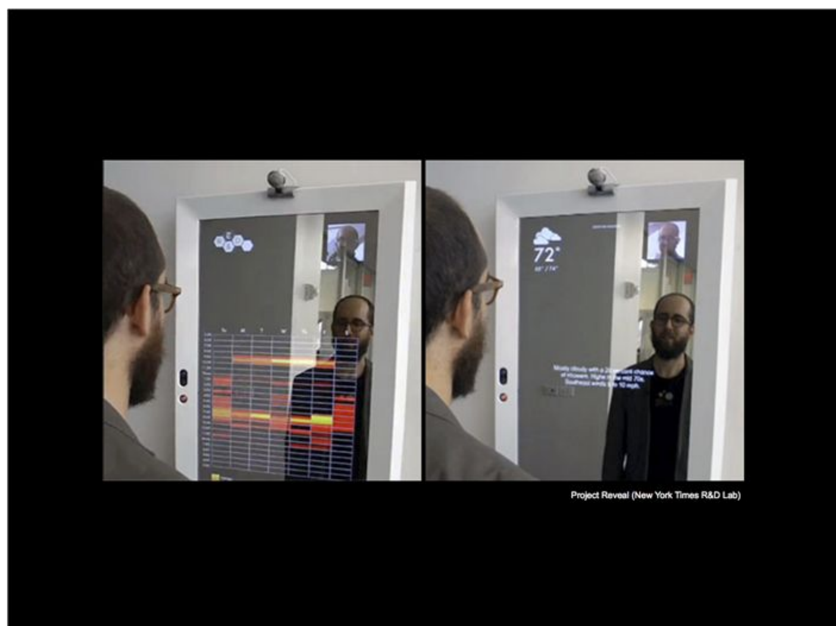


图9-4：《纽约时报》研发实验室的Project Reveal项目（http://nytlabs.com/projects/mirror.html）

Laura Kurgan是美国地理信息设计实验室（SIDL）的负责人，图9-5是他们设计的一个叫作“百万美元街区”的可视化项目。这个项目的数据来自谷歌的犯罪数据。Laura提取了监狱服刑人员的家庭住址，并计算出政府管理该服刑人员所支出的费用。因此图中的颜色深度代表了政府在管理该区域的服刑人员方面所支出的费用，颜色越红代表支出越高。某些街区的费用支出甚至高于100万美元。这个项目的意义在于，要想在地理信息图可视化上做出新意其实是不易的。这要求研究人员的可视化思路非常开阔，对数据的理解十分透彻。像图9-5这样的地理信息图就体现了不一样的可视化思路。

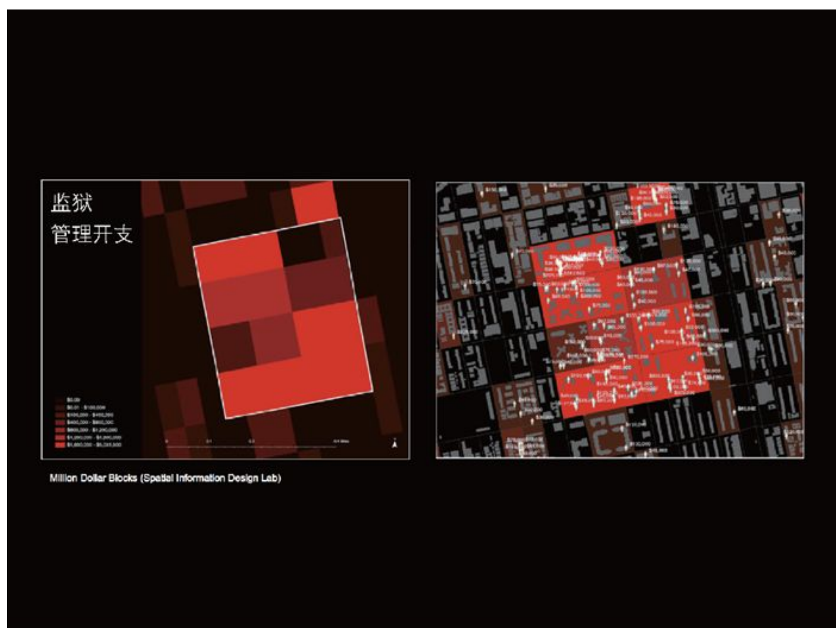


图9-5：地理信息设计实验室（SIDL）设计的“百万美元街区”项目（http://www.spatialinformationdesignlab.org/）

9.4 Mark的数据可视化项目

我们对Mark的个人影响以及他的数据科学哲学已经有了大致的了解。接下来我们来看一看Mark本人的一些可视化项目。

9.4.1 《纽约时报》大厅里的可视化：Moveable Type

Mark带我们参观了他和他多年的合作者媒体艺术家Ben Rubin一起完成的一项位于《纽约时报》总部大厅里的可视化项目（在完成这个项目之后，Mark就利用休假的一年时间去了《纽约时报》的研发实验室工作）。《纽约时报》曼哈顿中心区总部大楼位于纽约42街第八大道，这个可视化项目就展示在总部大楼的一楼大厅内，图9-6就是这个项目在展示时的情景。



图9-6：《纽约时报》总部大厅，Moveable Type可视化项目，作者：Ben Rubin和Mark Hansen

该项目使用了两面墙，每面墙280个文本显示盒子，共计560个。每个屏幕循环显示一系列设计好的情景，每个情景都包含一个由数据模型驱动的情景主题。

对于每个显示盒子上要展示的情景，其背后的数据都来自《纽约时报》的网站内容，包括新闻、博客以及搜索引擎结果等。对网站内容的解析使用了斯坦福大学的自然语言处理技术（http://nlp.stanford.edu/），该技术可以将文本内容图解化。这些图解化的文本内容经过数据模型被进一步情景化。该项目最初设计了总计15个情景，每个情景的数据模型都通过编程实现，因此修改也比较容易。YouTube上有关于Mark和Ben对于该项目的访谈视频，感兴趣的读者可以搜来看一看（http://goo.gl/yhCG69）。

举其中三个情景主题为例。一个情景是用文字波浪的形式展示对某篇文章文本的解析结果。每一篇文章都用一段简短的语句表示，不同的文章之间通过这样一段简短的语句就基本可以感知该文章的基本内容了。

另外一个情景不使用文字波浪模型，其背后的数据模型可以提取出更加简要的文本结果，通常只包含一个数值和一个名词的组合，比如“18只大猩猩”。这样的名词组合会投射到显示器上并展示出来。

第三个情景更加有趣，显示屏会显示一个个自动化的填字游戏，其填字的过程还伴随着铅笔划纸的响声，给人的感觉就是显示屏在自己完成一个又一个填字游戏，非常生动。

图9-7就是一种显示盒子的内部图，看起来非常复古。每个显示屏其实都包含一个独立的、运行着Python的Linux处理器，以及一个声卡用来模拟各种各样的声音，比如钟的滴答声、打字的声音、波浪的声音等。具体发出何种声音要取决于显示屏所要展示的情景主题。



图9-7 Moveable Type的显示盒子

9.4.2 屏幕上的生命：Cascade可视化项目

Mark接下来提到了Cascade项目，该项目是由Mark和他的合作者Jer Thorp共同设计完成的。Jer是《纽约时报》的数据艺术家，同时也在bit.ly兼职。Cascade的可视化对象是人们在Twitter上对《纽约时报》文章链接的分享行为。

该项目试图通过分析足够的数据，较为完整地展示《纽约时报》链接被浏览、编译（通过bit.ly）、分享、解译（也是通过bit.ly）和点击的过程。图9-8就是对这个过程的可视化结果，看起来和Tarde当初的建议不谋而合。

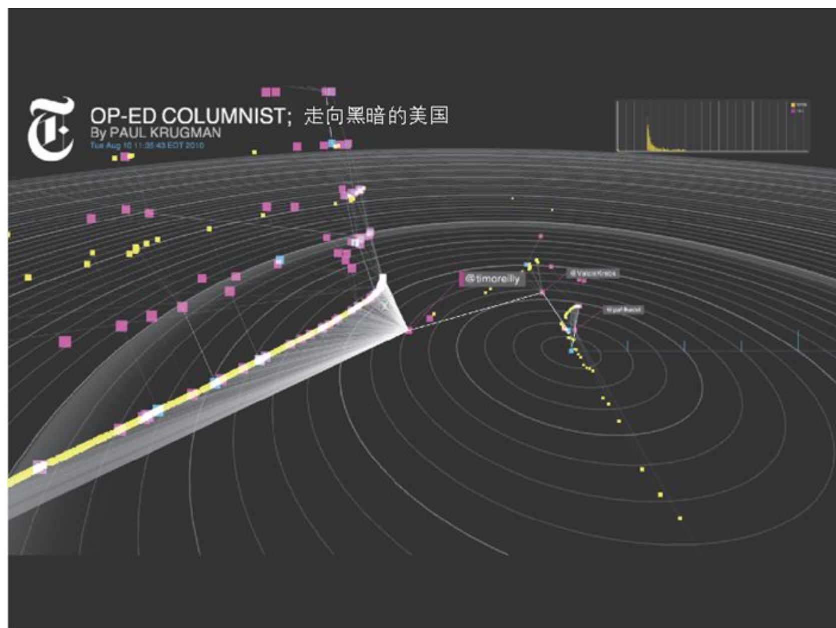


图9-8：Jer Thorp和Mark Hanse的作品：Cascade项目

对于Twitter的数据可视化，比较棘手的是对数据实际意义的解读。比如说，如果有17条推文都是关于同一条链接的，那么就很难确认到底哪一条“推文/链接”组合是实际上被人点击的。我们可以通过“猜”的方式，比如使用概率模型，根据时间戳信息大致猜出哪条组合被点击的可能性最高。另外，Twitter上的互粉关系也会被考虑进来，比如说，如果你的好友在你之前发布了同一条链接的推文，那么你的推文很可能只是对好友推文的转发。

对这个项目感兴趣的读者可以尝试去YouTube观看有关这个项目的视频介绍（http://goo.gl/uAOcg8）。



这个项目在两年前就已经完成了。在这过去的两年时间里，Twitter的规模已经发生了翻天覆地的变化。

9.4.3　Cronkite广场项目

另外一个可视化项目展示在位于得克萨斯州大学奥斯汀分校的Cronkite广场。该项目由Mark、Jer和Ben共同设计完成，它仍然与新闻内容有关，但是从图9-9中可以看出，文字被投射在了整个大楼的外部结构上。

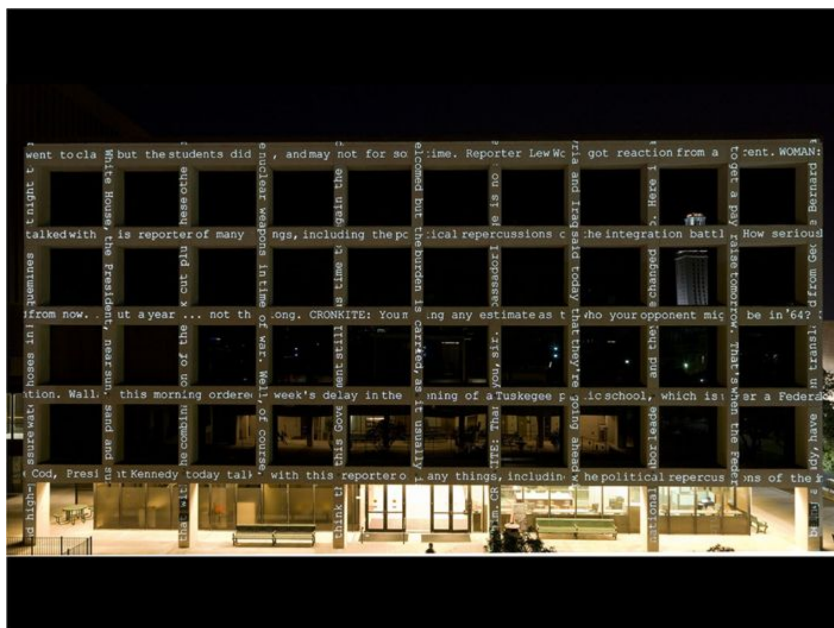


图9-9：Jer Thorp、Mark Hanse和Ben Rubin共同设计制作完成的项目：And That's The Way It Is（“就应该这样”）

在Cronkite广场上，每天晚上这些词语会被6个不同的投影仪投射在这个建筑物的外墙上。这些词语的来源包括Walter Cronkite的新闻联播以及一些当地的闭幕电台节目的新闻内容。词语是某条新闻内容的提炼，以句子的形式投射在外墙上。譬如，墙上会出现类似“她到底该

作何反应呢？”或者“你会养什么样的狗狗？”等，类似于新闻的标题，但是更加简约。

9.4.4 eBay与图书网购

Mark的另外一个项目是eBay网站上的图书网购数据的可视化，这个项目是他和Jer Thorp共同设计制作完成的。试想一下，如果让你把eBay网站上最近两年经由PayPal支付的图书网购数据用可视化图形的方式展现出来，你该如何下手呢？Mark和Jer为eBay设计了一个实体的可视化方案，并在2012年举办的两年一次的ZERO1大会上展示了他们的成果。图9-10就是这个项目在eBay大楼前展示的照片，显示的具体内容类似于图9-11。



图9-10：Jer Thorp和Mark Hanse设计：eBay的图书网购数据可视化项目

```
4106.00,CHINESE OLD SIGNED CERAMIC SHUDEI BONSAI POT KYUSU NR
712.50,Omron HEM-650 Wrist Blood Pressure Monitor with APS
499.99,Panasonic TC-P42X3 42" Viera Plasma HDTV
491.00,$500 Best Buy Gift Card for $491.00
372.23,Droid Incredible 2 (Verizon)
346.99,Hondo Chiquita Guitar
329.70,IMATION CD-RW 48 SPINDLES
324.25,1979 Camellias of Yunnan Souvenir Miniature Stamp sheet
287.00,DIESEL New $690 Leather Jacket Coat M
279.79,NEW CALLAWAY TOP FLITE COMPLETE MRH GOLF CLUBS SET BLUE
275.88,AMAZING LABRADORITE 925 SILVER CUFF BRACELET BA31
227.16,NEW Chefs Banquet Survival Emergency Food 330 servings
212.30,Authentic Prada Blue Bag
203.50,Superb Chinese Carved Jade Plaque 18th C.
189.00,NEW IDF Tactical Combat Vest with Removable Backpack
187.98,HP OfficeJet Pro 8500A PLUS AlO Printer   A910g    NEW
182.90,Cisco Small Business RV042 Dual WAN VPN Router -  RV042
174.95,PHILIPPE STARCK Black VEILED Stainless VEIL NEW! PH5022
173.23,14K. GOLD CHANDELIERS EARRINGS NATURAL BLUE TOPAZ GEMS
137.99,FUJI FINEPIX JZ300 10x 12MP DIGITAL CAMERA SILVER NEW
126.00,New Stock White/Ivory Wedding Dress Sz:6/8/10/12/14/16
124.64,Lot of 700 HP 02 Mixed Colors Empty Ink Tank Cartridges
116.40,Goddess of Wisdom 2001 Barbie Doll NIB Mint Condition
112.08,Littmann Cardiology III Stethescope
111.11,ALLEN EDMONDS PARK AVENUE OX BLACK 8 D MEDIUM $325
105.95,REPRODUCTION GERMAN WWII WOOL SOCKS SIZE 2 RING (9-10)
99.00,TOSHIBA SATELLITE L305-S5875 15.4" WXGA  LCD SCREEN
97.58,calvin klein men's slim fit plaid suit pants
90.00,Pear Shape Diamond
80.30,HAPPY CHLOE DUCK 2010 SWAROVSKI  RETIRED  #1041293
79.99,Cisco 2600 series 2611 2-port Ethernet Router CCNA CCNP
77.77,PRO BICYCLE MECHANICS XLC TOOL KIT 33pc BIKE REPAIR SET
74.90,NEW LG VX10000 VOYAGER QWERTY TXT MP3 PHONE VERIZON
69.94,Modern Jacquard Bedding Comforter Set Queen Black/Grey
67.49,pearl gameboy advanced SP metal case 18 gamesECT.
```

图9-11：eBay项目背后的真实数据

他们最后想出来的可视化方案非常具有独创性。首先，他们选了Arthur Miller的著作《推销员之死》（http://en.wikipedia.org/wiki/Death_of_a_Salesman）里的文本内容，并使用土耳其机器人（见第7章）在文本中选择可以在eBay上买到的东西，比如“椅子”“长笛”“桌子”等类似的商品名。

当收集到一串（10个左右）商品名后，再在当天的销售额数据中找出该商品的交易情况，并找到一些异常值，比如最大值和最小值。在检查完每一个商品名的销售情况之后，程序会在美国的邮编数据中挑选一个邮编号码，通常是一些小地方，比如蒙大拿州。

接下来书籍的网购数据被定位到蒙大拿州，在所有来自蒙大拿州的支付信息中随机挑选一本书，再从这本书里找可以买到的商品名，并重复上述过程。要直观地了解整个可视化过程是如何进行的，可以参考

网上提供的一段视频（https://vimeo.com/50146828）。

9.4.5 公共剧场里的“莎士比亚机”

最后一件作品来自于Mark、Rubin和Thorp，安装在美国公共剧院的大厅里。该项目是由37块LED片形灯组构成的一个穹顶型结构，被吊顶安装在大厅接待处的正上方，具体可参考图9-12。每一块片形灯组都对应于莎士比亚37部戏剧中的一部，灯组的长度越长代表该剧的长度越长。

因此，当你已进入大厅的时候，正对着你的就是莎士比亚作品中最长的一部戏剧作品《哈姆雷特》。



图9-12：“莎士比亚机”，作者：Mark、Jer和Ben

每片灯组负责展示相应戏剧的不同主题，输入的数据就是该剧所有的文本内容，而主题不同显示的内容也不同。比如说，某个主题只显示该剧正文中出现的某个名词，某个主题选择显示词组，而另外一个主题可能会选择显示一些剧中出现过的组合词。

注意，针对这些数据，这里的所谓数字人文通过MONK项目（http://monkproject.org）提供了极为丰富的XML（http://en.wikipedia.org/

wiki/XML）展示。

Mark说，由于是莎士比亚的缘故，因此不管用什么样的主题，这样的可视化方式都会显得很棒。但是，他们也在考虑如何通过这样的灯组更好地展现每部戏剧的特色：比如把字符看作符号，或者看作关键词或者演讲词等，再通过合适的组合方式更加多元化地展示每一部戏剧。

现在让我们回到Mark在最初向大家提出的一个最基本的问题：到底什么是数据呢？答案是：数据就是生活，它无处不在。

Mark在最后对如何获取数据给了一条很中肯的建议：做一个谦卑有礼的数据搜集者，因为人们总是更愿意将数据交给他们觉得更谦卑和有礼的人。

9.4.6　这些展览的目的是什么

这些展览既具有艺术和美学特征，又内容丰富，并且在设计时也避免了过重的说教性质。总体的设计原则是在美学和内容间找到一个平衡点。这些展览都具有故事性，不会让观众觉得乏味。由于各种编程和设计工具的高度整合和数字化，统计学家的作品看起来也越来越具有艺术性，设计师的作品也开始包含更多的数据和统计学元素。这样的整合趋势在刚才所展现的很多展览作品中，大家应该都深有感触。

9.5　数据科学和风险

接下来来自旧金山的Ian Wong将带来有关数据科学和风险管理的内容。Ian是Square公司的统计学家，他博士就读于斯坦福大学电子工程专业，主攻机器学习方向，但是他中途退学了（他获得过统计学和电子工程学的硕士学位）。在讲课的时候，Ian尚就职于Square，但他随后就从Square跳槽去了Prismatic，一家个性化新闻订阅服务商。

Ian在刚开始就给了大家三条心得。

- 机器学习不等于写**R**程序
  机器学习根植于数学，它的最终成果是通过编程实现的软件产品的有机组合。要做好机器学习，你必须具备相当的程序开发技能，能写一手漂亮的程序，且这些程序要具备良好的可读性和可实现性。程序的最终受众是广大用户而不是你自己，因此你应该确保程序经得起反复推敲，并能够实现产品化。

- 数据可视化不仅仅是一张好图
  对于一家优秀的数据产品公司来说，可视化应该成为产品开发文化的一个重要组成部分。好的数据产品应该有经过仔细构思的可视化模块，同时，好的可视化模块能够带给用户更好的产品体验。

- 机器学习与可视化推动着人工智能的发展
  人类本身的认知能力是十分有限的。然而，借助于数据和数据科学，我们有如披上了超人的外衣，得以在原本一片混沌的信息世界中自由翱翔。

9.5.1 关于Square公司

Square成立于2009年，其创始人是Jack Dorsey和Jim Mckelvey。公司在2011年有50名员工，到2013年已经发展到了500名员工。

Square的创始人觉得，现实生活中的交易都太过复杂，无论从支付方还是收款方来看，现有的支付方式似乎都显得过于烦琐。Square要解决的核心问题就是，如何使交易变得简单易行，以至于每个人都可以轻松地参与到商业活动中来？

Square给出了一种商业模式。首先收款方（商家）需要在Square上成为注册用户，下载Square开发的官方程序，之后Square公司会将一个信用卡读卡器邮寄给该商家。商家在收到读卡器之后，可以把读卡器和手机绑定（例如将读卡器插在手机的数据线接口上），打开Square程序即可接受支付。Square提供的小型读卡器为很多小型商家提供了一种最为迅捷的支付方式。在波特兰和旧金山，很多赶时髦的咖啡馆都开始采用Square的支付方式。对于消费者来说，所要做的只是拿出信用卡，在Square的读卡器上刷卡，最后在iPad上签上自己的名字。

当然，如果消费者（买方）成为Square的注册用户，也可以从中受益。你可以使用Square提供的电子钱包程序，这样当你在付款给Square的注册卖家时，甚至都不需要出示信用卡便可以直接支付，卖家所要做的只是在iPad的Square程序里点击你的名字。

固然，Square想为卖家提供最为迅捷的支付服务，但他们也同样需要做好电子支付的风险防范工作。下面我们就详细讨论一下Square支付模式所面临的风险以及相应的防范措施。

9.5.2 支付风险

Square的宗旨是为买方和卖方提供最为迅捷和自由的支付体验，然而这促生了一小批可能会滥用该服务的使用者。比如说，支付欺诈这样的违法行为不仅会严重损害用户体验，也是在挑战公司的价值底线。因此，如何打造一个稳健并且高度有效的风险管理系统是Square公司想要保持健康发展所要解决的核心问题。

那么到底如何才能有效地检测用户的支付操作并侦探出可能的欺诈行为呢？Ian解释说，他们在开发这样一套风险管理系统时，很好地将机器学习和数据可视化技术结合了起来。

机器学习在可疑支付行为侦测中的应用

到底什么样的支付行为是“可疑”的？这是我们要最先定义的概念。以下一些异常现象要引起我们的注意：同一时间内进行了大量小额支付、突然发生的大额交易或者是不同于平常交易频率的交易（不一致交易频率）等。

我们这里给出一个例子。比如John有一辆运送食材的卡车，在他开始物流业务的几周后，他在Square上的第一笔支付金额是1000美元。那么这样的一笔支付有没有可能是可疑支付呢？这很难直接看出来，因为John可能是有偿还能力的（可靠用户）。然而一旦John是一个支付欺诈者，那么所有的支付金额都将由Square自己买单。因此，Square所能赚取的利润与他们的风险管理水平有着直接关系，他们应该想尽办法尽量避免不可靠用户使用他们的支付服务。

但是反过来说，如果John有足够的偿付能力，而Square将他视作不可靠用户，这会在很大程度上影响Square的公司声誉和产品的用户体验，也会直接影响Square的利润率。因此，Square的风险管理系统既不能过于保守，也不能过于宽松；过于保守会失去很多可能的可靠用户，而过于宽松会让很多欺诈用户有机可乘。

这个例子很好地说明了Square公司在开发风险管理系统时所面临的两难境地，从机器学习的角度来说，Square需要同时考虑伪阳性和伪阴性的问题：伪阳性会损害用户体验，而伪阴性会让公司蒙受直接的经济损失。

从公司的角度来看，来自于买方和卖方的风险都需要考虑。本章主要考虑来自于卖方的风险，也就是收款方（比如John）可能带来的风险。

Square公司每天要处理的交易额高达数百万美元，因此这样一个风险

管理系统需要自动化，并且有足够快的反应速度。对每一笔交易和每一个用户，都能及时做出判断。那么这样一个系统到底应该如何实现呢？其背后的数据又是什么样子呢？Ian先展示了该系统中的三种数据形态，可见图9-13、图9-14和图9-15。

Payment
payment_id seller_id buyer_id amount success timestamp

图9-13：支付数据

Seller
seller_id sign_up_date business_name business_type business_location

图9-14：卖方数据

Settlement
ach_entry_id state timestamp

图9-15： 结算数据

这里一共涉及三种类型的数据：

- 第一种是支付数据，包含交易号、卖家识别号、买家识别号、

交易额、交易成功与否、以及时间戳等信息；

- 第二种是卖家数据，包括卖家识别号、卖家注册日期、卖家店铺名称、店铺类型、店铺位置等信息；
- 第三种是结算数据，包括结算识别号、结算发生地和时间戳信息等。

在Square上的每笔交易都会在一整天之后结算，因此风险管理系统对交易的风险度识别并不需要在分秒内完成。当然，处理的速度固然是越快越好。

图9-16是这个风险管理系统的交易处理流程图。每一项交易活动的具体信息都会经过风险模型的分析，完全不可疑的交易可以直接通过审查，可疑的交易会再经过一遍人工核查。人工核查会更加仔细，分析人员会就可疑的交易信息逐条分析以确保风险可控。在人工核查阶段确认可疑的交易会被冻结，该项交易会交给后续分析人员以作进一步的深度核查。所有确认安全的交易会进入结算阶段（图9-15的结算数据就来自于该阶段）。

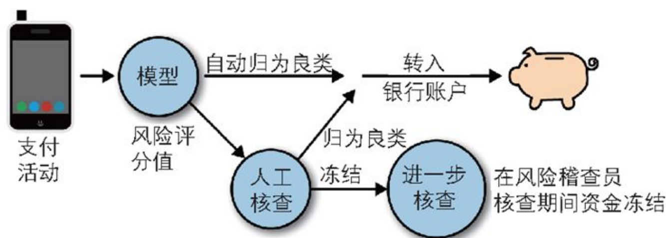


图9-16：风险管理模型的运行流程图

接下来的问题就是如何搭建图9-16中的风险模型。同所有机器学习模型一样，风险模型的输入值是有关交易的特征变量，输出值可以是一个二元值（1代表可疑，0代表不可疑）。因此从机器学习的角度来看，这是一个典型的监督式学习问题。但从操作层面来看，这个模型远远不止一个监督学习模型那么简单。从流程图中我们知道，模型的结果要比单纯的二元值复杂得多，有些交易会被标记为良性交易而直接通过，有些会被标记为恶性交易而直接否决，更多的是可疑交易，需要交给人工核查人员以便进一步确认其风险。具体有多少种标记方式要由风险管理团队讨论决定。

因此，严格意义上来说，这算是一个半监督学习模型，它具备监督学习模型的典型特征，也有一定程度的非监督学习特性。因为非监督模型的特性，到底应该有多少个标签是需要不断调整和改变的。但是需

要注意的是，在经过几个月的模型打磨之后，模型的非监督性质会逐渐消失，最后会成为一个严格意义上的监督性学习模型。Ian接下来要着重与大家分享的内容主要是有关监督性学习模型的。

众多周知，监督性学习模型大致包括以下几个步骤：

- 获取数据；
- 提取特征变量；
- 训练模型；
- 评估模型效果；
- 应用模型。

然而，在实际问题中合理地实现这些步骤也并非易事。就连上述步骤的顺序是否应该严格遵守都众说纷纭。Ian则建议大家在建模之前应该充分地分析问题，了解问题的实质并且明确分析的目标。也就是说，要在模型建立之前就想好如何评估模型的效果。

9.5.3　模型效果的评估问题

现在让我们集中精力关注一下模型效果的评估问题。Ian提到，人们大约会在以下三种情况下遇到麻烦。

定义误差指标

到底如何衡量模型的建模效果是十分重要的问题。之前我们讨论过的混淆矩阵，又称作真值功能表，是衡量分类模型效果的标准工具，如表9-1所示。

表**9-1**：实际值预测值列联表，也叫作混淆矩阵

	实际值 = 假	
[illegible]		
[illegible]		

最常见的模型指标是模型的准确度（Accuracy），根据表9-1中的符号表示，它的定义为：

$$准确度 = \frac{TP+TN}{TP+TN+FP+FN}$$

准确度可以解释为模型答对标签类型的概率。但是对于交易中的欺诈

行为来说，因为阳性值的总数偏少1， 因此准确度很难准确地衡量模型预测效果的好坏。这是因为即便假设所有的预测值都是阴性，模型也会有较高的准确度，因为大多数实际值都是阴性。但是很明显这个假设没有任何的预测价值，我们所关心的是阳性值。

> 1也就是说大多数交易还是属于正常交易。

在这种情况下，精确度（Precision）或者召回率（Recall）则能更好地体现模型的预测效果。精确度的定义如下：

$$\text{精确度} = \frac{TP}{TP+FP}$$

它的含义是指被预测为欺诈的交易事实上确实是欺诈交易的概率。

召回率的定义为：

$$\text{召回率} = \frac{TP}{TP+FN}$$

它的含义是事实上是欺诈的交易可以被模型侦测出来的概率。到底应该使用哪一个指标取决于很多因素。就成本因素来说，未侦测到的欺诈行为与侦测有误的交易所导致的成本，对于不同的数据和不同的数据问题，其区别是巨大的，而且前者的计算比较容易，后者属于潜在成本，很难准确计算。

定义标签

标签值对于监督性学习的重要性不言而喻，Ian认为标签信息是“被忽视的另一半数据”。在本科阶段的统计学课程和很多数据科学竞赛中，通常数据的标签信息都是给定的。但在实际数据中，标签信息是最难定义和捕捉的信息，但对于建模来说又是最为重要的信息。它不仅与我们的目标函数有关，它在很多情况下就是我们建模的目的所在。

对Square公司来说，定义好标签值是至关重要的一环，因为准确地定义标签值意味着准确地定义了以下概念。

- 什么是可疑的交易。
- 数据的“粒度”如何？是以事件（交易）为主体，还是以个体（用户）为主体，还是以两者为主体？

- 我们能否得到可靠的标签值？是否需要从外部数据中获取更多的信息以帮助做出判断？

最后Ian提到，标签值定义中的不确定性将给模型本身的预测精度带来很大影响，其中一种情况就是标签值的“高度失调”（比如之前提到的阳性数据量过少的情况）。

特征选择与模型学习过程中的诸多挑战

Ian说，特征变量的提取是对领域知识的抽象化和模型化。一旦机器学习模型搭建好并开始运行之后，基本上大量的精力会放在如何更好地抽象化领域知识、提取特征变量上：比如，根据模型的反馈，添加新的特征变量。然而，新变量也并非越多越好，因为每个变量的可学习程度是有限的。

当数据明显失衡的时候，要千万小心“过拟合”的出现。充分地学习好一个特征变量所要求的样本量，与模型所要预测的标签值数量是成正比的（在欺诈侦测的例子中，就是欺诈交易的样本量）。

当某个变量是分类型变量并包含较多类别时，其在模型中的应用要引起分析人员的额外注意。比如说“邮政编码”这样的分类变量可能有成百上千个不同的类别，但是在建模过程中，这样多类别的分类变量基本是无用的。需要通过组合的方式加以简化。Ian和他的团队甚至动用了模型，只为了减少某些分类变量的类别个数。

第二个问题与数据的稀疏度有关。对于新注册的卖家，这个问题显得尤为重要。新卖家的注册信息对于模型来说是未知的，而在建模之前又必须先做好特征的定义和提取。如果新卖家的注册内容包含很多新的信息，已有的特征指标系统就会显得过于保守和陈旧，不能很好地纳入新的信息。

最后的问题，也是机器学习中最头疼的问题之一：你需要根据反馈不断地调整模型，直到模型达到满意的预测效果。模型调整的范围十分宽泛，找到一个最佳模型无异于大海捞针。比如说，你需要决定是否考虑特征变量间的交互性，是否就“类别失衡”问题给予相应的调整；如果使用的是集成模型，你还需要确定一个合适的抽样框架。

另外一个棘手的问题是用户的刻意对抗行为。在电子商务中，一个恶意的用户会想方设法规避风险系统的欺诈侦测。一种典型的做法是，用户会注册十多个账户，每个账户所用的注册地址都有细微的差别，如果侦测系统主要在用户的注册地址上做文章，那么这样的做法对于

逃避侦测系统的检测会十分有效。对于风险管理系统的设计者来说，要严加防范用户的对抗行为。并且，由于这样对抗用户一直在学习如何规避系统的检测，因此系统的设计和管理人员也要不停地更新系统的核心模型。

> 关于变量的标签值
>
> 在加州大学伯克利分校信息学院举办的一年一度的DataEDGE会议上，Micheal Chui （任职于麦肯锡）与Itamar Rosenn（Facebook聘请的第一位数据科学家）讨论了如何更好地定义“重度用户”。这其实直接涉及标签的定义问题。判断一个用户是否是“重度用户”对于企业来说十分重要。但是什么样的用户可以定义为“重度”却没有统一的标准。事实上也不需要统一的标准，因为不同的公司在不同的数据问题上，对“重度”的定义可以不尽相同。但大体上，“重度用户”具有一些共有的行为特征，譬如他们访问服务的频率和次数很高，他们创建和更新日志的次数较多等。从某种意义上来说，这是一个半监督学习问题，因为标签值的定义和预测过程是在同时进行的。

9.5.4 建模小贴士

下面是一些有关如何更好地打造模型产品的贴士。

- 模型不同于黑匣子
  模型产品最好具有高度的可解释性，我们不能假设使用的模型是万能的，因此当模型对数据失效的时候，我们可以打开模型，看看里面到底发生了什么，以便更好地改进模型。

- 勇于尝试
  做模型和做科学实验是类似的，你需要不停地尝试，模型不会一蹴而就，也不会一劳永逸。当你觉得模型A和模型B都合理的时候，应该毫无犹豫地把两个模型都尝试一遍。当然，模型的尝试也有一个上限，你不能总是处在尝试阶段，产品总是要拿出来的。

- 模型和软件包不是万能的
  你也许经常会听到人们讨论“你用的是哪个模型”“你用哪个软件包”等，这样的人往往不知道他们在做什么。真正懂模型的人应该关注模型内部的细节，而不是模型本身，或者是使用什么样

的工具。

Ian同样也注意到，人们就某种算法经常会讨论使用哪个软件包的问题。比如，如果一群人都在用R做机器学习，他们可能在讨论`randomForest`、`gbm`、`glmnet`、`caret`、`ggplot2`、`rocr`等软件包。如果是一群用`scikit-learn`（Python的机器学习模块）的人，他们或许会讨论`RandomForestClassifier`或者`RidgeClassifier`等。但是，这样的讨论其实没有必要，软件包毕竟只是工具，真正重要的是模型背后的理论。

程序的可用性与可读性

对于数据科学家来说，编程是一项必备技能。对于写程序，Ian鼓励大家要注重程序的正确性、结构性、可用性和可读性。

如果你不使用别人编写好的软件包，而是自己写算法，那么代码的可读性和可扩展性会直接影响之后的产品质量。举个例子，在编写一个随机森林算法时，如果硬编码随机森林两大参数之中的任何一个，都会导致程序的后期扩展相当困难。如果有可能，参数要用软编码，这样不仅用户可以改变它的取值，后期的扩展和接口也会相对轻松很多。另外，不管你有多忙，时间有多赶，一定要写好测试程序。

在Square，为了保证程序的可用性和可读性，编程人员根据机器学习模型的特征将程序按照语义划分为不同的构件并存放在不同的文件夹中。大体包括以下5个构件。

- 模型
  核心算法部分

- 信号
  数据读取和变量生成

- 误差
  模型效果评估

- 实验
  一些探索性数据分析和实验等

- 测试
  所有的测试用代码

当项目上线之后，编程工作基本会集中在“实验”文件夹内。大量的新

数据会带来新的挑战，模型需要不断地改进，因此分析人员要不断地实验，做探索性分析，找到模型新的改进点。 这样做不仅能促使探索性分析带给分析人员新思维和新视角，也能尽量避免重复性劳动。对于团队来说，你可以看到同事都尝试了哪些分析，得到了哪些结论，这样在自己探索时，就可以“站在同事的肩膀上”了。Ian直言不讳地说道，在团队中工作，就是要写生产型的代码（production code）。

项目应该包含哪些构件和文件夹并没有成文规定，John Myles White对此有过讨论，可见这个Project Template项目（http://projecttemplate.net/）。如果你是R用户，Ian建议多去Github上看一看大牛的R项目的代码。如果有可能，你也可以尝试写一些软件包。Hadley Wickham的devtools（http://adv-r.had.co.nz/）包是广大R包开发者的福音。Ian说，要想写出漂亮的代码，与写出漂亮的公式证明十分相似：你需要大量认真的练习，从失败和用户的反馈中不断学习和改进。

作为课后练习，Ian建议大家比较一下R中的caret包（http://cran.r-project.org/web/packages/caret/index.html）和Python中的scikit-learn模块（https://github.com/scikit-learn/scikit-learn），看一看谁的源代码具有更好的扩展性和可读性？为什么？

找到小伙伴

编程是一门艺术，但也讲求团队协作。闭门造车往往事倍功半。就好比你学一门外语而没有人跟你交流一样，你永远也无法掌握这门语言的精髓。

在写程序的时候找到一个志同道合的小伙伴很重要，因为互相可以检查代码、发现问题并及时交流和改正。在Square，每一位工程师都有义务检查其他工程师写的代码。这不仅仅是为了改正代码中可能存在的错误，同时也保证了代码的质量，这包括代码的可用性和可读性等。

当你找到自己的编程小伙伴之后，还要找到一个你们共同感兴趣的问题，这样才能一起编程。买一套工作站并配上两套鼠标和键盘，接下来就可以一起协作了。你们会互相讨论如何解决面临的问题，将问题解析成块并一一攻破。这就好比汽车比赛，一个做驾驶员一个做副驾驶。驾驶员是真正操控车辆的人，也就是真正写代码的那个人，而副驾驶则负责检查代码并指示未来前进的方向。当驾驶员在写代码的时候，副驾驶应该不断地问自己“我能否看懂他写的这段代码”，或者“这段代码能够写得更简洁一些吗”等类似的问题。当副驾驶犯糊涂的时

候，应该及时与驾驶员沟通以消除疑惑。找到小伙伴的目的就是为了相互监督和学习，在学习中成长。

当然，驾驶员和副驾驶应该定期交换角色。如果你们合作得很默契，你会发现一切都变得十分高效。但是协作其实是一件费神费力的工作，刚开始人会很容易疲惫。但是随着默契度的增加和协作模式的完善，连续工作的时间会越来越长。

如果身边没有这样的小伙伴，你还可以使用git。熟悉git的工作流程，在拉拽请求（pull request）时要有建设性的意见和贡献。这和学术界的“同行评审”是一个道理。

将机器学习模型产品化

机器学习模型产品化的过程中有很多非常棘手的问题，如下。

1. 到底如何将一个模型“产品化”？

2. 如何实时地将实际数据转换成“特征变量”，并传递给模型？

3. 如何遵守“所见即所得”的原则？也就是说，如何最小化产品在线上和线下的表现差距？

在课堂上以及大多数机器学习竞赛中，模型的评估数据是静态的，对于模型的运行速度及模型运行的实时性没有要求。因此，人们总是倾向于使用复杂的模型，因为能够得到更好的预测效果。这导致大多数模型都非常臃肿，使用的特征变量都极尽复杂。很多数据科学家往往对此不以为意，因为他们认为似乎凡事都有解决办法，如下。

- 数据的维度太高？没关系！用奇异值分解就完事了。
- 数据是有关到达率的？没关系，让我们先对历史数据拟合一个泊松模型即可。
- 时间序列数据？用傅利叶变换吧！

然而，由于存在种种时间或者空间上的限制，我们在建模的时候并不能随心所欲。从产品的角度来说，模型需在瞬间做出反应并给出预测值。因此，要尽量避免模型被不必要地复杂化。

很多模型从理论上来看，其核心部分无非是特征变量之间的点积（比如广义线性模型和支持向量机模型等），或者是一系列有关特征变量的`if-else`关系语句（决策树模型）。因此，模型中最耗费时间的部分往往是特征变量的种种计算。计算方式有两种方式：批量式和实时

式。

由于模型的特征变量规模、复杂程度以及时延要求的不同，特征变量的计算方式也有所不同：可以采取纯粹的批量式或者实时式，也可以两种方式结合。具体采用何种方式，要具体问题具体处理。需要实时预测的情况，最好在模型的训练阶段采取批量式的计算方式，在预测阶段则采取实时式的计算。

MapReduce是一种常用的批量式计算框架。但由于Square苛刻的时延要求，Ian和他的机器学习团队在尝试开发一个实时的特征变量计算系统。

该系统的设计要保证历史特征变量和在线特征的计算方式绝对一致。换句话说，不论变量来自于训练阶段还是实时数据，都应使用相同的计算方式。只有这样，建模者才能确信模型可以正常工作。

9.6　数据可视化在Square

Ian接下来列举了Square的风险管理团队应用数据可视化完成的一些任务：

- 使交易的审核更加有效；
- 更好地展示单个客户的行为模式以及客户间的行为模式；
- 监测商业运作的健康度；
- 产品环境分析。

他们开发了一个检视用户行为的工作流程系统，该系统实时地展示用户的特征，包括用户的历史交易行为、地理位置信息、用户评价、联系方式等。该流程系统在很大程度上依赖于数据可视化技术。营运团队每天要处理大量的用户交易检视任务，如果没有软件工程师和数据科学家提供类似工作流程系统的帮助，他们每天加班到凌晨也无法完成任务。有了这样的可视化流程系统，营运团队的工作会变得十分轻松，可疑的交易行为和用户的侦测会变得更加容易。Ian觉得，机器学习和数据可视化的有效结合，能够碰撞出很多思维和技术的火花，这是一个数据公司所求之不得的。

Square的办公室里安装了很多电视机，上面显示着公司数以千万计的客户们的实时状态。大家亲切地称呼这些电视机为“信息发射机”。这种可视化与欺诈侦测并没有直接的关系，却能够把公司运行的实时趋势和状态展现给员工，这可以极大地激发大家的工作热情。

这样的可视化与“产品环境分析”的概念很像，它直观地提供了与产品间接相关的因素的运行状态，这对开发产品本身是大有裨益的。它有助于我们对于产品直接相关的数据的认识更加透彻和全方位。有些产品环境甚至可以重点可视化，Square的风险管理团队就针对很多产品环境因素开发了单独的可视化界面。

Ian的团队甚至可视化了一批风险管理指标，其中包括可疑交易每天的“清理率”和“冻结率”、每天处理的可疑交易数等。整个指标可视化系统看起来十分高端，Ian甚至可以直接看出哪些分析人员处理得可疑案例最多、平均每个账户的处理时间是多少、哪些因素会增加核查的负担等。

总而言之，Square是可视化技术的忠实拥趸，用户的大部分行为都被纳入到了可视化管理系统当中。可视化分析俨然成为了Square公司“产品环境分析”的重要一环，发挥着重要的作用。尤其对于Ian的风险管理团队来说，他们尤其相信，“看得清才能行得远”。

Ian最后讲到了自己的数据科学背景以及从事这个行业的一些建议。首先，他形象地说，如果要画出自己的技能水平与数据科学背景的关系图可以用R函数`plot(skill_level ~ attributes | ian)`，那么技能水平应该取对数。这是因为学习一项技能通常并不需要太久的时间，但是要精通该项技能却需要长时间坚持不懈地练习。 他同样认为生产力的测度同样应该取对数，对于那些引领同行的软件开发者们来说，它们开发一个软件包的速度要远远快于其他开发者。

作为临别赠言，Ian鼓励大家：

- 要玩就玩真实的数据；
- 在学校里好好地学数学、统计学和计算机；
- 多去公司实习；
- 保持一颗好奇的心。

9.7　Ian的思维实验

假设现实生活发生的每条交易的数据都可以被记录和保存下来，你应该怎么使用这样的数据？

9.8　关于数据可视化

并不是每个人都能做出类似Mark所展示的那些可视化项目，但是掌握

一些可视化技术对提高我们的数据分析技术以及交流能力会有如虎添翼的作用。同数据科学一样，数据可视化本身也是一样综合性的技术，要想真正掌握并成为大师绝非易事。为了方便大学进一步学习可视化技术，下面列出一些我们认为比较经典的书籍或者资料。

- 关于数据可视化的一些基石性的技术，Micheal Dubokov的介绍很不错，可参考http://www.targetprocess.com/articles/visual-encoding.html。

- Nathan Yau是Mark Hanse在UCLA的博士研究生。他在http://flowingdata.com/发表了很多关于R中可视化技术的博客和文章。Nathan也写过两本有关数据可视化的书：第一本叫作*Visulize This: The Flowing Data Guide to Design, Visualization, and Statistics*，由Wiley出版社出版；另一本叫作*Data Points: Visualization That Means Something*，同样由Wiley出版。

- Scott Murray写过一系列关于d3的教程，链接地址为：http://alignedleft.com/tutorials/d3/。由O'Reilly出版的*Interactive Data Visualization*就是基于这套教程。

- ggplot2的作者Hadley Wickham（ggplot2是R中基于Wilkinson的作图语法的作图系统）的书*ggplot2: Elegant Graphics for Data Analysis*（Springer出版社的Use R！丛书系列之一）。

- 数据可视化的经典教材还包括Edward Rufte（统计学家，被认为是数据可视化之父，Mark Hansen和他是不同年代的人）的*The Visual Display of Quantitative Information*。该书的重点在于可视化原则和可用工具。第1章我们提到的William Cleveland有两本关于可视化的著作：*Elements of Graphing Data*（Hobart Press出版社）和*Visualizing Data*（Hobart Press出版）。

- O'Reilly出版社的一些新书也非常不错，比如*R Graphics Cookbook*、*Beautiful Data*和*Beautiful Visualization*。

- 我们认为数据可视化不能只局限于对工具和统计学的学习。许多艺术院校都会出版关于设计的理论书籍，其他有关于新闻学原则和心理学的内容也与可视化密不可分，了解这些知识对于设计更好的可视化产品都很有好处。

- Jeff Heer是斯坦福大学的教授，d3的作者之一（另一位作者是Micheal Bostock，他之前任职于Square公司，后来跳槽去了

《纽约时报》）。他十分推介BretVictor的讲座“Describing Dynamic Visualizations”（动态可视化简介）。Jeff说Bret为大家展示了数据可视化一个崭新的视角。

- 如果有可能，你要与艺术家或者平面设计师合作。

数据可视化练习作业

选修这门课的学生与正在读这本书的你一样，知识背景差异很大。如果你觉得自己是可视化方面的新手，Rachel建议你一定要看一看Nathan Yau的博客网站，从中选取一两个可视化题目自己动手做一做。在动手的同时，要勤思考，也许你的一些想法会在某些方面改善Nathan Yau的可视化结果。

对于课程中其他具备可视化背景的学生，我们建议他们参加Hubway的数据可视化竞赛（http://hubwaydatachallenge.org/）。Hubway是波士顿市的自行车共享项目。为了更好地宣传这个项目，主办者主动公开了项目的数据集，并组织了这样一场可视化的竞赛。虽然竞赛已经结束，但是数据还是公开的。如此有意思的数据，是练习可视化的大好机会。Rachel班级里的两位学生，Eurry Kim与Kaz Sakamoton赢得了该项赛事的“最佳叙事性奖”（best data narrarive），Rachel非常引以为豪。图9-17就是他们的竞赛作品，他们利用可视化技术，用一种浪漫温暖的方式讲述了一个关于自行车共享的故事。

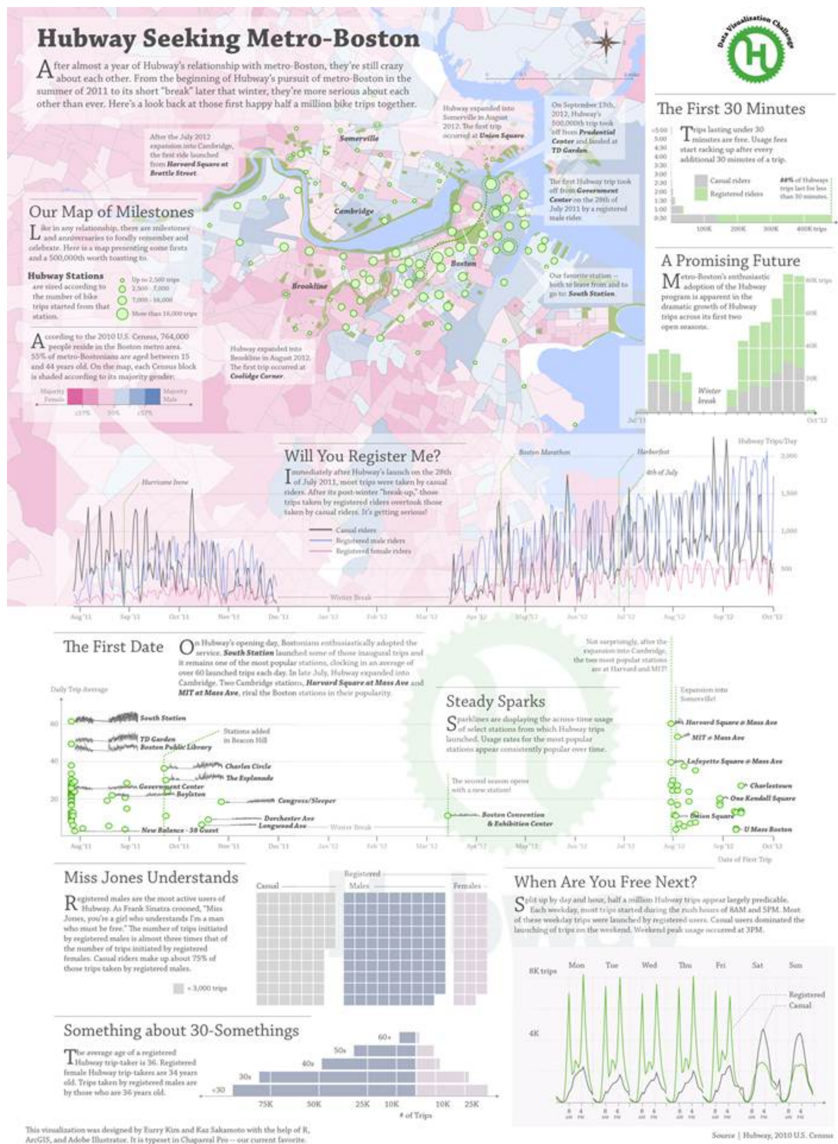


图9-17：Eurry Kim和Kaz Sakamoto参加Hubway公共自行车项目可视化竞赛的最终作品，以及这个项目在波士顿中心区实施的情况

第 10 章　社交网络与数据新闻学

本章要讨论的是过去5到10年间学术界和工业界讨论的两大热门主题：社交网络分析和数据新闻学。社交网络本身不是一个新的概念（这里的社交网络并不只局限于在线社交网络），对它的研究早在10年前就已经风生水起了。它既是社会学的一大研究对象，也吸引了计算科学家、数学家和统计学家的眼球。统计学中的图论就与社交网络的研究密不可分。随着在线类社交网络服务的迅猛发展，　如Facebook、LinkedIn、Twitter和Google+等，社交网络研究的可用数据如潮水般涌来，为社会科学的定量研究提供了全新的研究材料。

Morningside Analytics是一家专注于社交网络数据研究的公司，我们首先会听一听来自他们的声音，并且简要回顾一下有关社交网络的基本理论知识。社交网络数据中蕴含着丰富的故事资源，这是另外一种形式的新闻：数据新闻。我们之前说过，数据科学家的背景理论知识构成就像人的基因组一样，有很多成分：包括数学、统计学、可视化、编程等。然而从事社交网络与数据新闻学研究的科学家所需要具备的知识构成与传统的数据科学家有着细微的差别。虽然两者都要求具备发现问题，用数据解决问题并与人交流发现成果的能力，但数据新闻学家却具备一些独特的研究视角。来自O'Reilly的编辑Jon Bruner会与我们分享他有关数据新闻学的一些想法和理念。

10.1　Morning Analytics与社交网络

此篇的贡献者是来自于Morning Analytics的科学家John Kelly，他将与我们分享有关社交网络分析的心得。

Kelly1990年本科毕业于哥伦比亚大学，之后在哥伦比亚大学的新闻学院相继获得了硕士和博士学位。他的研究领域是社交网络分析与政治统计学。他还抽空在斯坦福大学选修了问卷设计、博弈论等课程。他的硕士论文是与Marc Smith在微软一起完成的（http://videolectures.net/marc_smith/）。论文的题目是有关社交网络与政治讨论的联动关系的研究。在上大学之前，Kelly主要从事音效设计方面的工作。他曾经在哥伦比亚大学艺术学院数字媒体部门做过三年的主管。当年他在越南与妻子度假的时候自学了Perl和Python，然后就成为了一名程序员。

Kelly觉得，若想在这个领域做好自己感兴趣的事情，就必须具备一些

数学、统计学和计算科学方面的知识。这就好比大厨一样，要想做好一桌好菜，没有锅碗瓢盆再好的厨艺也无处施展。

Kelly感兴趣的是人们为什么会在社交网络上聚集起来？他们在社交网络上都在做些什么事情，这些事情对于政治和公共政策的可能影响是什么，等等。他所任职的公司Morning Analytics有很多来自政治组织智库的客户。这些客户找到Morning Analytics的目的很简单：了解有关社交网络与媒体活动的趋势及其对政治活动的可能影响。

面对这样的客户，良好的交流与展示技巧是十分关键的。可视化在这其中往往起着至关重要的作用。John不仅需要具备扎实的理论功底和分析能力，还需要熟练地掌握可视化工具，将分析结果和结论用直观的形式展现给客户。因为客户付钱的原因，并不在于你的分析发现了什么问题，而是你的发现对他们的决策有没有支撑作用。

案例-属性数据与社交网络数据

传统模型所针对的数据通常叫作案例-属性数据，每一条案例都对应某个人或者某个事件，相应的人或者事件又由很多属性构成。这样的模型早在19世纪30年代就被广泛地应用于市场调查研究中了，并且很快蔓延到市场研究的其他领域，包括政治研究。

Kelly指出，案例-属性这样的数据形式给分析带来了巨大的偏差和倾向性。即便这样的数据形式非常适用于现有的数据库系统，收集起来也相对容易，但却极大地束缚了数据分析和研究的广度和深度。

Kelly提到了另外两位来自欧洲的社交网络分析领域的开拓者，Paul Lazarsfeld和Elihu Katz。他们认为社交网络分析不仅要考虑到个人，更要注重人与人之间关系的研究，这是社交网络的本质特征。

社交网络数据分析与案例-属性数据分析没有绝对的孰优孰劣，然而有时候应用前者可能会带来更好的效果。比如说，联邦政府想知道人们对于出兵阿富汗的看法，通常的办法是花很多钱做大规模的民意调查，得到的数据是典型的案例-属性数据。然而，Kelly指出，民意不仅仅是单个个体意见的线性组合，要找到整个社交网络中有影响力的组织或者个人，他们的意见往往左右着群众的意见。这样的分析只能通过社交网络分析来实现，传统的案例-属性数据分析则会束手无力。

试想一下，如果现在让你回到1750年的欧洲，调查民意并判断未来的政治走向，你会怎么做呢？如果你学过社交网络分析的话，你应该直接分析上层社会的联姻关系，而不是跑到大街上发问卷。

其实在很多情况下，人们习惯于使用案例-属性数据是因为这样的数据比较容易收集、储存和分析；即便对于很多问题可能根本就不适用。

Kelly想要告诉大家的是，现实世界是一张复杂的网络而不是一群独立的个体堆砌而成的。社交网络分析在很多问题上都要优于传统的案例-属性数据分析。

10.2　社交网络分析

图论和社会关系计量学对社交网络分析的形成和发展起到了推动作用。欧拉用图论的方法巧妙地解决了Konigsberg的七桥问题，而社会关系计量学的出现则得益于20世纪70年代计算机技术的迅速发展，大型数据的处理从不可能变为现实，其创始人为Jacob Moreno。

哥伦比亚大学的退休教授Harrison White和社会学家Robert Merton是社交网络分析的鼻祖。他们认为，对人类行为的分析不能只局限于研究个体的属性，人们之间互相影响所形成的网络（或系统）也应该成为研究的主要对象。

然而到底如何研究这个网络似乎并不是一件很容易的事。Kelly指出，网络的分析无非是对微观个体和宏观整体的整合性的研究，也就是说既要研究局部个体也要照顾大局整体。

在实际生活中，个体与总体总是通过各种途径联系起来。比如，人们的购买行为形成一张消费网，而人们的竞选和投票行为则构成了一张政治网络。我们亟需合适的工具解析这一张张复杂的网络，这也就是社交网络分析的终极使命。

10.3　关于社交网络分析的相关术语

网络中的基本单位称作参与者（actor）或者节点（node），可以用来表示人、网站或者其他你能想到的一切事物。这些节点通常在网络图中表示为一个个实心点。节点之间的相互连接的关系称作关系连接（relational tie）或者边（edge）。比如说在网络图中，如果你对某人点了赞或者和某人互粉，你们之间的关系就可以用一条边连接起来。被边连接的两个个体称作“二分体”（dyad），如果三个节点相互连接则称作“三分体”（triad）。比如说，节点A与节点B之间有边，节点B和节点C也有边，则ABC三个节点形成“三分体”的必要条件是A和C之间也有边。

一张网络如果太大，则有必要着眼研究其中的子群体（subgroup），也叫作子网络，这包括子群体中的所有节点以及它们之间的边。子网络与母网络从形式上来看并没有本质区别，只是规模大小有别。

如果两个节点之间有关系，则从表现形式上来看，它们之间必有线相连。从网络连接的角度来说，对某人点赞与实际生活中与某人住在一起没有形式上的差别，都可以用一条边连接起来。一个社交网络就是所有节点与边的集合。

一个社交网络可以很简单也可以很复杂。最简单的网络类似于Facebook上的好友网，两个人之间要么是好友要么不是，任何两个人之间都可以成为好友。

稍微复杂一点的是二分图（bipartite graph），节点只存在于两个不相交的子集当中，同一个子集的节点之间无边相连。举例来说，一个子集可以是人，而另外一个子集可以是公司，二分图描述的既不是人与人之间的关系，也不是公司与公司之间的关系，而是人与公司之间的关系。如果某个人在某家公司的董事会任职，则他们之间可以用边连接起来。这样的例子举不胜数，比如一个子集仍然是人，而另外一个子集可能是人们感兴趣的社会活动。那么二分图所描述的关系，既不是人与人之间的关系，也不是社会活动之间的联系，而是人与社会活动之间的对应关系。如果某个人对某项社会活动感兴趣，他们之间则存在一条边。

最后一种网络叫作“自我中心网络”（ego network），顾名思义，就是围绕一个节点有很多连接线的网络。以Facebook为例，一个人的好友圈就是以这个人为中心的自我中心网络。研究表明，一个人自我中心网络的规模和复杂程度与他的社会经济地位有着直接的联系。通过观察他的自我中心网络就能大致了解他所拥有的社会地位。

10.3.1　如何衡量向心性

给定一个社交网络，人们会问的第一个问题往往是：谁在这个网络中的重要性最大？

要回答这个问题，首先需要定义何为重要。重要性在社交网络中又叫“向心性”，这里我们简单介绍几个常用的向心性测度指标，并给出相应的例子。

第一个测度叫作自由度，这通常指的是在网络中有多少人与你有边相连。以Facebook为例，自由度就是你的好友个数。

第二个测度叫作“紧密度”。如果你的好友与你的连接越紧密，你们的“紧密度”就越高。

为了更好地定义向心性，我们需要定义连通图中两个节点间的“距离”。尤其是对于两个非直接相连的节点，距离的定义非常重要。假设节点x和y之间的距离用$d(x, y)$表示，最常见的距离可以定义为节点x和y之间的最短路径的长度。于是，节点x本身的“紧密度”可以定义为：

$$C(x) = \sum 2^{-d(x,y)}$$

接下来介绍另外一种向心性的测度指标，叫作“中间性”（betweenness），它衡量的是在你的网络中，你的好友之间的紧密程度。也就是说，他们之间的最短路径是否通过你的节点，如果通过，说明你在连接这两位好友过程中起到了直接的作用。如果你的中间性指标很高，则代表你对好友的影响很大，起到了信息中转站的作用。

为了精确地定义中间性的概念，我们假设节点x和节点y之间的最短路径数为$\sigma_{x,y}$，最短路径中通过节点ν的路径数为$\sigma_{x,y}(\nu)$，那么节点ν的“中间性”定义为：

$$B(\nu) = \sum \frac{\sigma_{x,y}(\nu)}{\sigma_{x,y}}$$

上式的和取自所有与ν不同的节点（x, y）组合。最后一种向心性测度指标叫作“特征值向心度”，我们会在10.6.1节详细介绍。直观的解释就是，如果你与更多的名人们有联系，那么你的特征值向心度要高于常人。谷歌的PageRank理论就是这种向心性测度的一个很好的应用。

10.3.2　使用哪种向心性测度

介绍了这么多的测度指标，对于到底该使用哪个指标则是一个头疼的问题。总有一些自称是“权威人士”或者“专家”的人说应该使用某某指标，然而事实上根本没有统一的标准。不同的问题，不同的网络类型，应该使用的指标都不尽相同。

举个例子来说，设想我们要在穆斯林兄弟会成员的博客中找到一个有影响力的博客。一个可能的做法是，先列出网络上最有影响力的前100个博客，以自上而下的顺序找到其中与穆斯林兄弟会有关的博

客。这样的做法其实完全跟我们想要解决的问题没有关系。你可能找到的是一个与穆斯林兄弟会毫无关系的人，他只是偶尔写了一篇与穆斯林兄弟会有关的博客而已。这个例子告诉我们，如果我们关注的对象只是一小群人，那么我们自然要把视野转到这一小群人所形成的子网络中，而不是在茫茫大网中寻找。

另外需要提示大家的是，根据问题的不同，测度指标的选取也会有所差异。比如说在找博客的时候我们使用的测度指标，如果放在Twitter的数据上就可能完全不适用。

一个值得注意的问题是，一些不良用户如果知道了系统使用的是何种测度指标，他们会想方设法提高自己的知名度。做法很简单，如果Twitter的系统使用的是特征值向心度，那么不良用户可能会注册多达5000个账户，每个账户之间都互相关注，然后再策略性的关注某一个特定账户，这个特定账户的知名度就可能会大大增加。这样的情况肯定是Twitter所不愿意并想极力避免的。

如果你用Python，那么NeworkX（http://networkx.github.io/）或者igraph（http://igraph.sourceforge.net/）可以用来计算这些测度指标，在R中可以使用statnet包（http://statnet.org/）。如果你喜欢用Excel，可以选择NodeXL（http://research.microsoft.com/en-us/projects/nodexl/）。另外我们注意到，斯坦福大学的Jure Leskovev正在研发一个基于C语言的网络分析包，感兴趣的读者可以关注一下（http://stanford.io/18Pejdt）。

10.4　思维实验

假设你是来自华盛顿某智库的研究人员，手头有1000万美元的预算。你的任务是预测埃及这个国家未来的政治走向：包括哪个政党可能执政，埃及在5年、10年以及20年之后会变成一个什么样的国家。你所拥有的数据十分广泛，包括埃及公民的教育信息、通话和短信记录、家庭住址、所有政治组织和公司的网络使用情况，以及所有公民的Facebook和Twitter账户信息等。

在你觉得如何使用这些数据之前，你还应该注意这些信息并不是一成不变的。随着时间的推移，有人会注销Facebook，政治组织可能会转为地下党（意味着他们的信息会很难获取）等。即便大多数都有Facebook账户，但还是有少部分人从来不使用它，这一少部分很可能是你最感兴趣的那部分人。从这个角度来说，通话和短信记录数据可能更加有用。

你可能觉得我们假设的情况很不现实，野心太大。其实不然，德国当年就通过出口西门子制造的宽带设备给伊朗，成功地掌控了伊朗整个国家的宽带使用数据。这样的例子在国家与国家之间经常发生，比如美国就曾经帮助过巴基斯坦，俄罗斯也曾经帮助过叙利亚完成过类似的“间谍”工作。

对于这样一个问题，我们需要改变我们思考问题的方式。面对数据，人们习惯会问：我能从这个数据中得到什么结论呢？这样的想法会严重束缚我们分析和思考问题的广度和深度。这种从数据出发的态度是不可取的。相反，我们应该问类似这样的问题：预测一个国家的政治走向到底要预测哪些具体的指标？我们需要哪些数据支撑我们的分析？这些问题不是以数据为绝对导向的，而是更有深度，也更能带给分析人员有意思的结果。

总之，我们应该以问题为导向，而不是被数据牵着鼻子走。先把问题设定好，再找数据并从数据中找到信息。

10.5 Morningside Analytics

Kelly在课堂上展示了一张世界上最大的14个博客圈的网络地图。要理解这些网络地图，可以想象有一股类似于风的力量将节点（代表博客）往图的边缘推动，而恰好有一股反作用力将这些节点以某种固定形态拉在了一起。这股反作用力就是博客之间的连接关系。图10-1表示的就是其中的阿拉伯博客圈。

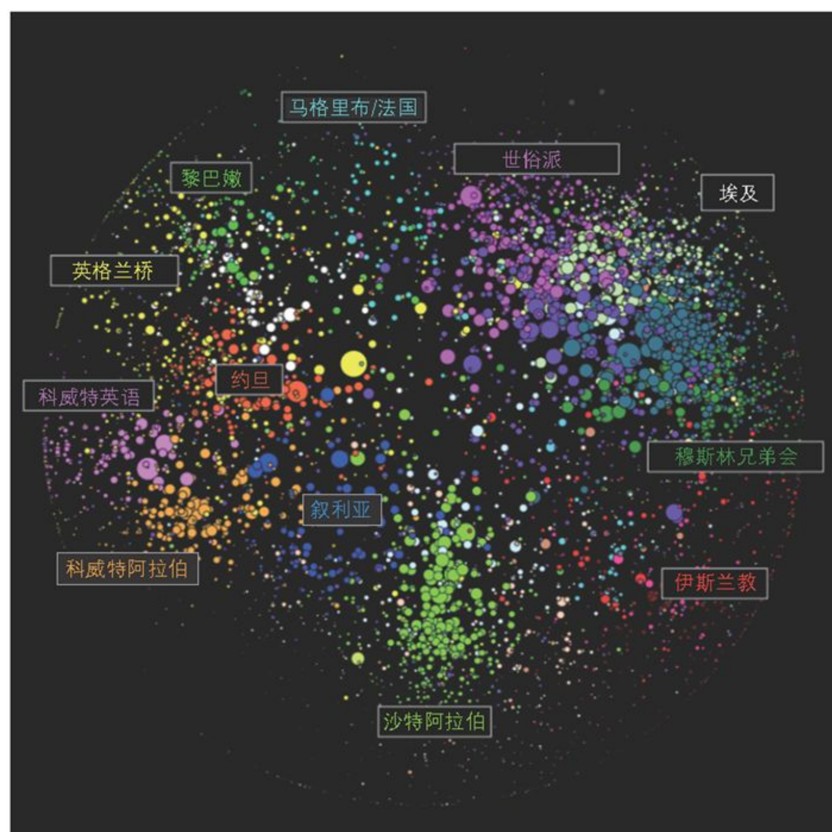


图10-1：阿拉伯博客圈

图中每个点都代表一个博客，不用的颜色代表不同的国家。点的大小取决于相应博客的向心度的大小，越大则代表着点的直径越大。向心度大代表该博客的连接线越长。这样的网状结构图有助于我们深入了解和洞悉博客圈的情况。

如果只分析每个博客圈中的文章，最典型的方法是用自然语言处理技术（NLP）分析其中的文本，那或许我们会失去很多更有价值的分析对象：博客和博客圈的相互关系。举例来说，应用社交网络分析技术，我们可能会发现不同博客圈的不同关注点，这是纯文本分析技术的盲区。

这样的博客圈可以想象成是某种高维复杂关系在二维平面上的投影。它们到底有什么不同要取决于它们原本内部的复杂关系。从形式上来看，我们似乎可以随便地添加颜色、设定圆圈的大小等。文本分析技

术可以在这里帮助我们确认我们分析的结果是否是有意义的，我们所要做的就是如何更好地、定性地解释这些网络。

比如说，法国的博客圈可能讨论的主题是美食，而德国的博客圈主题可能与政治和一些奇奇怪怪的嗜好密不可分，而英国呢？Cathy插嘴说道：英国的两大博客主题应该是成人片和同性恋成人片。应用网络分析，我们可以将这些国家的博客圈主题联系到这些国家的保守性与自由度。

由于俄罗斯严格的网络审查制度，他们的博客圈连看起来与其他国家的有着显著的区别。

这里使用的分类算法是Fruchterman-Reigngold算法，该算法可以将一些有共同影响的博客圈在一起，其分类的结果具有优良的可解释性。图10-2就是英语圈博客的分类图。

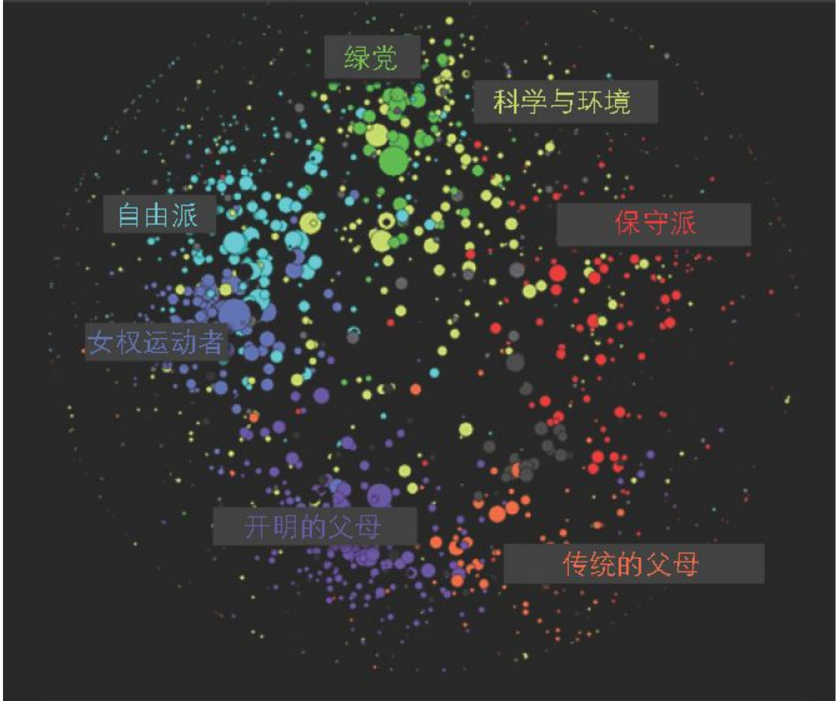


图10-2：英语圈博客分类图

可视化与中观视角

社会媒体公司要么有数据要么有技术。这些技术基本是一些申请了专利的情感分析引擎或者是一些其他的高精尖技术。也就是说，我们完全不知道这些技术背后的细节，因此要想真正理解你要分析的对象，在使用他们提供的技术的同时，你还需要知道如何使用可视化技术把分析的结果具体化、形象化，以转变成直观的解释。

例子：如果你想要分析大选的结果，你更想知道广大人民群众的看法。因此可以搜索有关“妈妈”或者“体育迷”的博客，因为这些人是老百姓的可能性更大。如果你搜索一些党员的博客，那么他们会写什么内容不看都可以猜出个八九不离十。

Kelly举了另外一个关于大选的例子。比如说马丁·路德金的“我有一个梦想”的演讲视频和罗姆尼的竞选视频，在大选期间被博客圈分享的模式是显然不同的。前者会出现在博客圈的各个角落，而后者只会集中出现在一些支持共和党的保守博客圈内。

但是，如果从视频被分享的次数的直方图来看，似乎后者的分享次数会更多。因此如果不用网络分析的方法而只关注统计量的话，我们很难发现后者其实是人为操作的结果。

Kelly也为哈佛大学的Berkman互联网与社会研究中心工作。她之前在2008年和2011年分析过伊朗的博客圈。分析结果显示，2008年和2011年的博客圈特征没有显著区别：年轻的反政府民主主义者、诗人（是伊朗社会的重要组成部分）和政府保守派是博客圈的主导群体。

然而，分析的结果同样显示，在2008年到2011年的三年间，只有15%的博客圈没有改变过。

这个例子告诉我们，即便个体的改变是巨大的，但却对整体没有显著影响。也就是说，像社交网络分析这样注重群体网络研究的方式，会比案例-属性数据那样注重个体研究的方式带给研究人员更深层次的洞察力。

对于国家和社会这样宏观的研究对象来说，社交网络分析是更加适用的方法：它带给我们是一种介于宏观与微观的独特视角，我们称为中观视角。

10.6 从统计学的角度看社交网络分析

从统计学的角度来看可以将网络看作一个随机数或者随机变量，是由某个随机过程或者某个概率分布形成的。因此，网络也可以生成样本。如果现实生活中的网络是某个随机过程或者概率分布的样本实现，那么我们会问一些统计学中的典型问题。比如说，Twitter的网络有哪些典型特征？这个网络能很好地代表总体的特征吗？

社交网络分析中的很多问题都来源于数学、统计学、计算机科学、物理学和社会学等学科。它的应用则不仅仅局限于这些学科，甚至可以运用到fMRI图像研究、流行病学、社交网络研究（如Facebook和Google+）研究等。

10.6.1 网络的表示方法与特征值向心度

网络节点间的连接方式有两种：有向连接和无向连接。例如在Twitter上，如果我关注了你而你没有关注我，这就是一个单向的连接，它是有向的。某些网络是无向的，从Twitter的角度来说，就是要么我们互相认识，要么我们完全不认识。

一个包含N个节点的无向网络可以用一个$N * N$的二元矩阵表示，矩阵中只有0和1两个值。如果第（i, j)个元素是1则代表从节点i到节点j是相连的。这样的矩阵也叫作相邻矩阵（adjacency matrix）或者关联矩阵（incidence matrix）。这样的矩阵也可以用来定义有向网络，而无向网络的特别之处在于这样的矩阵是对称的。

另外一种表示网络的方法是使用列表和多元列表。比如说，对于一个节点i，可以用一个列表列出连接i的所有的边，这样的列表也叫作关联列表。这样做的好处是，节点可以有多重属性，而且这样的表示方式可以节省很多存储空间。如果节点有多重属性，那么相应的表示方法叫作多重列表。比如说节点代表人，那么可能的属性包括人的性别、年龄、身高等。如果节点是人的某种行为，习惯或者爱好，那么相应的属性也会随着改变。

连接边也可以有自己的属性，比如可以被赋予权重代表该连接的强度。这种情况下，网络同样可以表示为一个$N*N$的矩阵，只不过应该把0和1换成相应边的实际权重值。

如果用相邻矩阵表示一个网络，那么就可以定义特征值向心度（10.3.1节）了。假设相邻矩阵为A，该矩阵的特征值为λ，对应的特

征向量为x，由线性代数理论我们知道：

$$Ax = \lambda x$$

其中：

$$\lambda_i / > 0, i = 1, \cdots, N$$0, i = 1, \cdots, N">

通过解$det(A - tI)$1便可以求出解特征值和特征向量。通常的做法是将特征值从大到小排列，并取其中最大的特征值及其对应的特征向量。特征值代表的就是向心度的大小，而对应的特征向量则是向心度在各个连接上的得分值。某个特征值λ对应的特征向量x的求法是解下面的齐次方程式：

> 1这也称作特征多项式。

$$(A - \lambda I)x = 0$$

这里得到的特征向量x就是我们想要的特征向量向心度。

到目前为止，这些公式只是告诉我们怎么计算得到特征向量向心度，却没有告诉我们为什么是这样。上面的叙述完全是线性代数求解特征值的理论，至于为什么特征向量可以用作向心度的测度指标以及相关的证明和例子， 可以参考相关特征值和特征向量的资料。感兴趣的读者可以参考这篇文章：http://goo.gl/UVkLoF。

如果你不喜欢从线性代数的角度计算特征值，也可以用下面一种迭代的方法得到具有最大特征值的特征向量。在迭代之前，假设一个长度为N的向量，其元素是每个节点的自由度，通常这些自由度已经被标准化，因此整个向量的元素之和为1。这个初始向量所包含的信息与节点之间的连接程度没有任何关系。为了得到向心度，对于某一个节点，在下一步迭代的时候将其近邻节点的自由度加总在该节点上，以此类推给所有其他的节点。在一次迭代之后还要进行一次标准化操作以保证整个向量的元素之和始终为1。反复迭代，每一次迭代的近邻规模都增加一个节点，这样最后得到的向量就是近似为特征值向心度的向量。上面那篇文章也给出了该方法的理论推导。

10.6.2 随机网络的第一个例子：Erdos-Renyi模型

之前已经说过，我们可以将网络看作由某个随机过程产生的样本。具体来说，可以认为网络的连接边的分布是来自于某个分布函数，并假

设边之间是相互独立的。

因此，如果有N个节点，则一个有$D = \binom{N}{2}$种可能的节点组合（也叫作二分体），每个节点组合之间都有可能有一条连接，或者没有。因此一共有2^D种可能的网络。对于每一条连接边，最简单的情形是假设每条边的存在都服从一个参数为p的伯努利分布，由此得到的网络模型也叫作**Erdos-Renyi**模型。

> 伯努利网络
>
> 在伯努利假设下，观测到一个所有节点都相互连接的网络的概率为p^D，而所有节点都不相互连接的概率为$(1-p)^D$。这两种网络代表了两种极端，现实中的网络基本都介于两者之间。Erdos-Renyi模型与伯努利网络是同义词。从数学上来看，这样的模型只具有理论研究意义，通常被用作证明更加复杂模型的某种性质。

10.6.3　随机网络的第二个例子：指数随机网络图模型

由于假设条件太不现实，因此伯努利模型很难在现实生活找到可应用的例子。比如说，最常见的好友网络，或者学术界学者之间的合作网都具有明显的传递性（transitivity，也就是说，如果A认识B，B认识C，那么A也认识C）、聚类性（clustering，有相同特征或者兴趣的人倾向于一类，从网络形式上来看，就是在母网内有很多抱团的小网络）、相互性（reciprocity或者mutuality，也就是说如果A加了B为好友，那么B也会加A为好友）、中间性（betweenness，通常是有影响力的节点才具备此特征，它在网络的信息流动中扮演着重要中间人的角色）。所有这样特性的存在都说明我们需要更加复杂的模型。

类似于上面的网络特性都可以用数学语言表示。比如说，传递性就可以表示为网络中的三角形的个数。

指数随机网络图模型（ERGM）是社会学中广泛使用的网络模型，它可以涵盖大多数我们讨论过的网络特性。

ERGM的研究对象是网络中的某些典型变量：比如网络中三角形的个数、连接边的个数、双星的个数（双星指的是某个节点有两个连接边，因此如果一个节点的自由度为3，则它包含3个双星）。这些变量用z_i表示，并且假设其分布的参数为θ_i。例如，用z_1表示网络中三角形的个数，并且该变量的参数为θ_1。如果θ_1是一个较大的正数，则代

表该网络中有较多的三角形，其节点之间具有较强的传递性。

还有一些较为复杂的变量，比如k星（k-star，与双星类似，k星表示某个节点具有k个连接边，一个自由度为$k+1$的节点有$k+1$个k星结构）。一个较为复杂的ERGM模型可以表示为：

Invalid Equation

该模型的含义是：对于一个网络Y，出现我们观测到的网络y的概率可以表示一系列网络变量的线性组合形式。

从形式上来看，伯努利网络是ERGM的特殊形式，也是最简单的形式。伯努利网络的变量就是网络中边数。

ERGM的推断问题

从统计学的角度来说，理想化但非常不现实的情况是，对于某个网络Y，我们可以观测到一系列的样本网络，$Y_1,...,Y_n$，其中n是样本量。每一个样本网络都假设有N个节点，并可以用一个相邻矩阵表示。

给定这些样本网络，我们假设它们是相互独立的，并来自于同一个概率分布模型。在这样的假设下，ERGM的推断是可能的。**2**

> 2也就是说，其中的参数是可估计的。

以伯努利网络为例，假设某条连接边存在的概率为p，那么观测到某一个特定样本网络集合的似然概率可以表示为：

$$\prod_{i}^{n} p^{d_i}(1-p)^{(D-d_i)}$$

其中d_i是第i个样本网络中观测到的连接边的个数， 而D是网络中所有二分体的个数。由此，p的估计值为：

$$\hat{p} = \frac{\sum_{i=1}^{n} d_i}{nD}$$

然后，从实际情况来看，我们不可能观测到一系列的样本网络，通常只有一个样本，也就是说有效样本量为**1**。我们用1个样本估计了模型

中的参数，这看起来似乎不可思议。对于伯努利网络模型来说，p的估计值就是样本网络中连接边的个数与二分体总是的比例值。这看起来很合理。

但对于更加复杂的ERGM模型来说，一个样本对于参数估计来说是远远不够的。当然，我们可以使用类似伪似然估计（pseudo-likelihood estimation procedure）这样的估计方法。但是即便是这样，还是会有很多困难。Mark Handcock 2003年的论文“Assessing Degeneracy of Statistical Models of Social Networks”（“论社交网络统计模型的退化性”,参见http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.81.5086）详细地讨论了该问题。他指出，即便是使用基于模拟的方法（比如**MCMC**，马尔科夫链蒙特卡洛方法），也很难避免“推断退化”问题（inferential degeneracy）。也就是说，所估计出来的网络很可能是一个退化的网络（完全相连或者完全没有连接的网络），或者说估计的结果没有一致性等很多问题。

关于随机图模型的其他例子：隐空间模型与小世界模型

由于指数随机网络模型中可能出现的模型退化以及模型估计不稳定的问题，研究人员提出了一种新的图模型：隐空间模型（latent space model）。Peter Hoff的文章“Latent Space Approaches to Social Network Analysis”（“基于隐空间模型的社交网络分析”，参见http://1.usa.gov/GzAT1Z）是该研究的开山之作。

隐空间模型主要解决一个问题：网络中有很多因素是不可观测的，这样的不可观测因素既是客观存在的，也是非常重要的 。传统的模型不考虑它们的存在，隐空间模型可以很好地捕捉到这些因素。比如说，Facebook上的好友网络，我们观测到的是人们之间虚拟的好友关系，但是至于这些人住哪、他们到底为什么会成为好友，单单从网络上来看是无从知晓的。

Watts和Strogatz于1998年提出了小世界模型（参见http://en.wikipedia.org/wiki/Watts_and_Strogatz_model），该模型要估计的网络介于完全随机网络与完全不随机网络之间。它想在现实网络世界中验证六维自由度理论。然而，人们对此模型的批评主要集中于它的过度均匀假设。因为现实世界的网络是没有尺度特性的，并且非均匀的。

除了上述模型之外，还有很多其他的图模型：比如，马尔科夫随机场模型（Markov random field）、随机块模型（stochastic block model）、混合会员模型（mixed membership model）、随机块混合

会员模型（stochastic block mixed membership model）等。关于随机块混合会员模型，可参考Edoardo Airoli等人的论文“Mixed Membership Stochastic Block Models”（“随机块混合会员模型”，参见http://dl.acm.org/citation.cfm?id＝1442798）。

下面列出一些关于社交网络分析的书籍，感兴趣的同学有时间可以读一读：

- *Networks, Crowds, and Markets*（《网络、人群与市场》，原英文版由剑桥大学出版社出版），作者是来自于康奈尔大学计算科学学院的David Easley与Jon Kleinberg；
- *Mining Massive Datasets*（《大数据挖掘》，原英文版由剑桥出版社出版）一书中关于社交网络图模型分析的章节，作者是来自斯坦福大学计算科学学院的Anand Rajaraman、Jeff Ullman和Jure Leskovec；
- *Statictical Analysis of Network Data*（《网络数据的统计分析》，原英文版由Springer出版社出版），作者是来自波士顿大学的Eric D. Kolazcyk。

10.7　数据新闻学

来自O'Reilly的Jon Bruner与我们分享了有关数据新闻学的内容。Jon之前在福布斯杂志做数据编辑，他的数据科学技能十分全面，他的研究和出版内容也都与数据相关。

10.7.1　关于数据新闻学的历史回顾

数据新闻学已经发展了相当长的时间，比如基于Excel的自动报告系统就是某种形式的数据新闻学。但是这样的自动报告系统一直是Excel一统天下，即使在今天，如果你能写一手不错的Excel程序，也有人愿意花大笔钱雇你。

API的出现以及计算机价格的平民化改变了这个现状，各种形式的自动化报告工具如雨后春笋般地出现。现在的数据分析工作，即便数据较大，一个人也可以单枪匹马地在个人笔记本上完成。越来越多的人开始具备基本的编程素养，甚至有些作家都可以写一手漂亮的数据分析程序。一些英语专业的学生也会花很多时间研究计算机和编程。更有意思的是，计算机出身的人也可以写出一手很好的文章。从趋势上来看，人们都变得越来越全面了。

在《纽约时报》这样的大型新闻出版机构，数据新闻学被细分为很多微部门：平面设计、交互设计、数据库工程、爬虫工程、软件设计、领域专家和写手，等等。有些人只负责提出问题，而真正动手干的又是另一批人。比如，Charles Duhigg任职于《纽约时报》，他最近收到了纽约州议会关于信息自由法案（FOIA）的问案，因为他是这个领域的专家，因此他清楚地明白就该方案应该提哪些相关的问题，但是他不会着手于具体的数据分析。分析的活是留给另一帮人干的。

如果是在一个小公司里，情况就大不同了。在《纽约时报》的大楼里，数据新闻部门有差不多1000个员工，而在《经济学家》杂志社，他们有130人，《福布斯杂志》有接近80个人。如果你在一个小公司工作，那么很可能所有的脏活累活你都得一个人全揽下：你要想该问哪些问题、该如何收集数据、自己做数据分析、自己写数据分析报告。当然，如果有两三个同事帮忙最好，因为每项工作都是技术活。

10.7.2　数据新闻报告的写作：来自专家的建议

Jon毕业于芝加哥大学，主修专业为数学。毕业后加入了《福布斯》杂志社从事写作工作。由于工作的要求，他也慢慢地开始做一些定量分析的工作。在报道亿万富翁或者政治家的社会贡献时，他也时常会用到一些图分析的工具。

在课堂下，Jon以自己的数据科学背景为例，为大家解释了“数据新闻学”的含义。

首先，数据新闻学需要大量的数据可视化工作，因为可视化是直观地报道和解释数据的最有效的工具。计算科学方面的知识对于精通数据新闻学也十分重要。因为时间就是生命，数据新闻要求对分析的工具掌握娴熟，能够熟练快速地处理好原始数据。数据新闻工作者要面对形形色色的数据，这就要求他们能够熟练地使用像Python这样的工具处理原始数据。Jon自己就精通JavaScript、Python、SQL和MongoDB。

统计学对数据新闻学的重要性同样不言而喻。统计学是数据分析的基石，是我们思考的方式。比如，某篇报道中可能会写“Twitter上平均每个女性有250个好友”，这里使用的是平均值。如果用中位数，则平均每个女性的好友数为0。因为原始数据是严重不对称的，使用不同的统计量则可以讲出不同的故事。

Brune说他本人在机器学习方面是一个不折不扣的新手，但是掌握一些机器学习的理论和技术对于数据新闻学同样重要。这也与你工作单

位的规模有关系，如果你任职于政府部门或者大型的日报集团，那么你需要是某个领域的专家。然而，如果你在一个小单位工作，就像我们之前说的，你什么都得懂。

另外两项对于数据新闻工作者来说至关重要的技能是：交流与展示。如何把一个个复杂的故事以通俗、容易理解的方式展示给读者，是数据新闻工作的基本要求。同样，数据新闻工作者要时刻准备着回答读者提出的各种问题，把问题转化成数据分析任务，再将分析的结果以同样通俗和容易理解的方式反馈给读者。

Jon给大家的最后一条建议是：时刻准备好改变主意！数据新闻工作类似于探索性数据分析，我们要自己做分析也要和领域专家打交道。信息会以一种非常难以预料的方式砸向我们，因此我们应该时刻做好改变主意、改变思考和分析方向的准备。

第 11 章　因果关系研究

到目前为止，本书讨论的模型和一些实例都是针对预测问题的。比方说，第8章我们介绍了如何预测人们对某件事物的偏好：比如一部电影或一本书。这样的模型可以纳入成百上千个特征变量，再利用变量选择的方法筛选出对于因变量最为重要的那些特征变量。模型的终极目标是最大化模型的预测准确度。在模型优化过程当中，变量本身的含义和解释就显得无足轻重了。尤其是当模型中的变量个数很多时，我们不可能逐一地解释每个变量的含义。

也就是说，如果建模的目的是最大化模型的预测精度，那么你大可不必花很多心思在变量的解释上。例如，一个亚马逊的图书推荐模型可能包含这样一个变量，即“你是否读过Wes McKinney的O'Reilly系列书*Python for Data Analysis*”，这个变量对于预测你是否会读这本书当然有用。但是，是否读过这本书就代表你会买下它？这可说不定。而且这个解释本身听起来就像是一段同义反复。当我们的建模标准是预测准确度时，我们可以这么做，不必担心如何理解或者解释变量间可能存在的因果关系。但是，如果你真的想要构建研究因果关系的模型时，就不能这么干了。

实际生活中不是所有的问题都是预测问题，你可能真的想要研究变量之间的因果关系。到底什么是因果关系？说白了，如果你想要做出某种行为导致了某个结果的论断，这便是因果关系推断。因果关系模型并不是一套完全不同于预测模型的统计方法。恰好相反，它其实是根植于传统预测模型（如逻辑回归、线性回归）的框架内的。但是，你的思路和目标就不再是优化模型以提高预测的准确性了，而是尽力分离出变量之间的因果关系。

这一章我们就要着重探讨因果关系的建模。我们特别邀请了这个领域的两位专家：Ori Stitelman和David Madigan。Madigan也是下一章的主要贡献者（我们会在下一章详细介绍他）。但是想要想理解下一章的内容需要仔细研读本章。Ori是Wells Fargo的一名数据科学家，他之前在一家法律事务咨询事务所工作，随后在加州大学伯克利分校获得了生物统计学的博士学位。他的主要工作是从数据中攫取信息、找出故事，并与相关领域的专家交流。在与形形色色的数据库打过交道之后，他提出了“数据直觉”这一概念。

11.1 相关性并不代表因果关系

确定两个变量间的因果关系是统计学的一大难题。想一想，当你说一个事件会导致另一事件的发生的时候，需要多大的勇气？实话来说，确定因果关系确实是一项艰巨的任务！

假设我们发现了冰激凌的销售额与泳衣销售额之间存在相关性，图11-1画出了它们的时间序列图，图中可以看到它们之间有明显的相关关系。

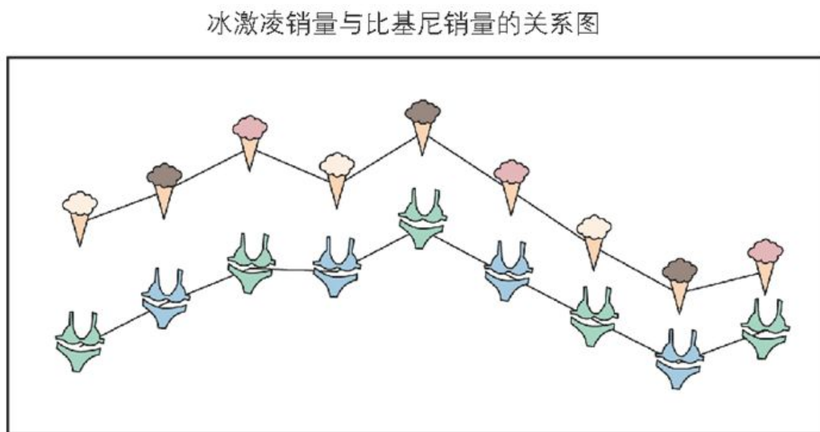


图11-1：冰淇淋销售额与泳衣销售额的时间序列图

虽然该图显示二者密切相关，但这并没有建立起任何的因果关系。现在重新考虑二者的关系，许多解释都可能符合图中二者的关系：也许是当人们穿着泳衣时，就特别想吃冰激凌？也许是每次吃冰激凌时，人们都会换上泳衣？又或者是我们没考虑到的第三个变量（如高温）导致上述二者同时发生？这些听起来都有可能，尤其是第三个关于高温的推测。因果关系推断就是为了理解在哪些情况下，变量之间的这种相关关系可以确定为因果关系。

11.1.1 对因果关系提问

对因果关系最自然的提问方式是：x对y的影响是什么？

来看几个例子：“广告对消费者行为的影响是什么”“药物对杀死某病毒有效吗”，又或者更一般的说法，“实验对结果的影响是什么”。



“实验”与“非实验”这两个术语来自生物统计学、医学与临床学领域，指的是患者是否接受了某种医学治疗或者处理。关于医学（流行病学）的例子会在下一章详细讨论。本章之后的讨论会经常使用“实验”和“处理”这样的术语。这些术语也经常出现在统计学和社会学研究的文献中。

实话说，因果推断中的参数估计是非常困难的。比如说，广告到底有没有作用？它的作用有多大？这是一个典型的因果推断问题，但是却基本不可能有精确的答案。因为其中因果关系的强度实在是太难估量了。人们通常花大力气研究那些简单易测的变量，但这些变量却并未能测出他们想要的东西，而大家不管三七二十一，都根据这些变量的研究结果做出决策，这样的研究是非常不负责任的。比如，营销人员会因为销售业绩好而受到公司的奖励，因为公司认为他们的营销努力为公司带来了更高的销售额。这是一个典型的因果关系推断，但是其中一个值得怀疑的地方是，销售业绩好可能是因为那些消费者本来就有强烈的购物欲望，跟营销人员的工作没有关系。这里面就有一个“干扰因子”的问题，它是因果关系推断的核心概念，下面我们就用一个更加详细的例子解释这个概念。

11.1.2　干扰因子：一个关于在线约会网站的例子

让我们来看这样一个例子，是有关一个叫Frank的寂寞的家伙在网上约会的事。假设Frank在一个约会网站上发现了心仪的对象。为了说服她出来跟他约会，他首先要写一封能引起她兴趣的邮件。他该在这封搭讪邮件里说些什么？如果Frank看过这位姑娘的头像，觉得她长得很漂亮。那么他能直接在搭讪的邮件里夸赞对方长得漂亮吗？也就是说，在搭讪邮件里就直接夸赞对方漂亮对Frank有好处吗？对方会买账吗？

理论上来说，Frank可以做一个随机实验。假设他心仪的对象有很多，这些对象构成一个样本。随机实验的做法是，将样本随机分成两半，一半的样本Frank会在搭讪邮件中称赞说她们很漂亮，而对另一半的人不做任何夸赞。如果前一半的对象反馈要明显好于后一半，那么就可以确认这样的搭讪技巧是有效的。

然而，不管是什么原因，Frank并没有这么做。这个做法听起来就挺疯狂的。于是得由我们来决定，对Frank而言，这样的搭讪风格是否有效。Frank能否搭讪成功现在完全取决于我们。

让我们先将这个因果问题明确地提出来：Frank在搭讪邮件中告诉一位姑娘她很漂亮会对他的搭讪成功率产生什么影响？换句话说，这里的“实验”或者“处理”，是Frank通过搭讪邮件告诉一位姑娘她很漂亮；而“结果”是这个姑娘有无积极的回复。这里的“控制实验”就是Frank在搭讪邮件中没有提到“漂亮”的事，而是扯了一些别的。



在这个例子中，其实还有很多因素没有考虑。例如，我们并没有提到Frank的为人。他也许是个怪胎，很不讨人喜欢。因此无论他说什么都没有姑娘愿意跟他约会。又或许他压根不会写“漂亮”这个词。相反，如果他是个帅哥、暖男或者名人，则不管他说不说，女方可能都愿意与他约会？另外，考虑到大部分的约会网站给男性与女性联系对方提供了同等便利的条件，有些姑娘可能会主动找上Frank，无论Frank有没有事先给她们发邮件。因此，从这些因素来看，因果推断其实是非常复杂的。

11.2 OK Cupid的发现

OK Cupid是一家在线约会网站，他们利用近50万会员的数据，分析了一些常用词和短语在第一次邮件（搭讪邮件）接触的时候对回复率的影响。分析的结果可见图11-2。

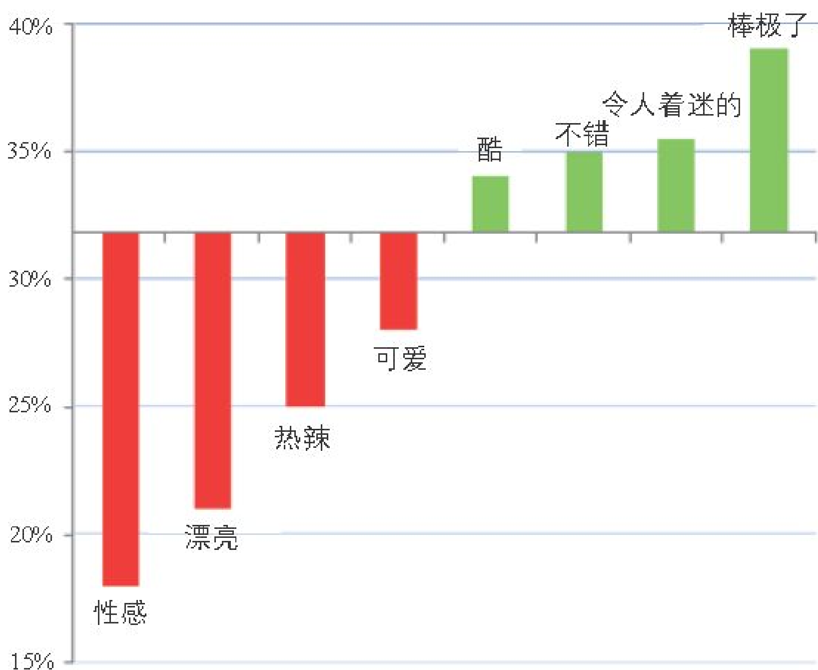


图11-2：OK Cupid的研究发现，在第一次接触性对话中使用“漂亮”一词不利于得到积极的答复

Y轴表示回复率。平均来看，所有邮件的回复率约为32%。然后，他们将这些邮件按关键词，如“漂亮”或“惊艳”来分类，并观察各类邮件的回复率。如果用条件概率来表达上述结果，可以说他们估计的结果是：P(回复) ＝ 0.32，而P(回复 | “漂亮”) ＝ 0.22。



上图遗漏了一个重要信息：那就是每个分类样本的的样本量。有多少人在首次接触性对话中使用了这些词？这个问题并不会改变所得到的结果，但它却有助于我们弄清楚图中那条位于32%的横线是否是根据每个分类样本的样本量计算的加权平均值。

他们把上述发现总结为搭讪的第一原则：“避免过度恭维。”他们还把这条发现发在了公司的博客上，题为“在线约会应该如何搭讪”。文中

说到："你也许认为人们都喜欢被'光彩照人''漂亮'及'性感'这样的词语包围，但从在线约会搭讪的数据分析来看倒并非如此。在见面之前，用这些词语搭讪往往会事与愿违。另外，当你告诉一位女士她很漂亮的时候，很可能是你不够帅。"

从统计学的角度来说，上面的例子叫作观察性研究。观察性研究指的是数据的生成过程没有受人为干扰，是自然生成的。这与人工设计的实验正好相反——在实验中各种因素都被人为控制，以研究某一个特定因素对实验结果的影响。从观察性研究的角度来说，能否根据上图就推断，在邮件中使用"惊艳"（fasinating）可以提高回复率，而使用"漂亮"（beautiful）就会降低回复率呢？

在回答这个问题之前，先考虑以下三方面的问题。

首先，在第一次搭讪的时候用"漂亮"这个词，往往在暗示对方自己也很帅？而对方可能会认为此人太过多情。另外，"漂亮"不仅可以用来形容女性，也同样可以用来形容食物、衣物等。因此，也要考虑词语使用的具体语境。

其次，上面的两点在因果推断的时候都要考虑到。但是即便语境本身很重要，但是从Frank个人的角度来看，其实关系并不大，因为不管他多么多情，或者他用"漂亮"赞美的是其他食物，这对于Frank搭讪的对象来说都是出自Frank之手，因此对于Frank本人来说，这些因素并没有干扰因果推断本身。

最后，要考虑的最重要的问题是，收到包含"漂亮"一词邮件的人很可能比较特别，她可能因为头像比较好看而经常被人搭讪。因此，这部分女性每天会收到成堆的邮件，而只有精力回复其中很少一部分，因此像Frank这样的人收到她们回复的可能性就更小了。

事实上，如果"漂亮"可以被完整地定义，它在这个例子中可被视作一个干扰因子。也就是说，如果这个女士真的"漂亮"，会同时影响到Frank是否给她发邮件以及她是否会回复Frank。当一个变量同时影响到"实验"本身，以及"实验"的结果时，它就是一个干扰因子。

如果考虑干扰因子的影响，OK Cupid的研究以及他们对上图的解释可能是完全错误的。但由于我们没法得到实际的数据，也不能妄下定论。但是，我们可以讲清楚我们需要什么样的数据，以及如何合理地分析数据。他们的分析和解释也许是对的，但单从一张图很难做出合理的因果推断。

11.3 黄金准则：随机化临床实验

我们到底应该怎么做才能确定变量之间的因果关系呢？

确立因果关系的黄金准则是使用随机化实验。顾名思义，随机化实验的关键在于随机化：样本被随机化为两个子样本，一个作为实验组（接受处理），另一个作为控制组。随机化之后，两组样本的表现差异就可以视作是“处理”因素引起的。从统计学角度来看，随机化保证了两个子样本都是来自同一个总体的同质样本，因此对于两个子样本来说，潜在干扰因子的可能影响是同等的。这从理论上排除了所有潜在干扰因子的影响。

随机实验的效果很好，因为在随机化的过程中，所有可能成为干扰因子的因素都被排除了（比如是否有吸烟史）。随机化保证了有抽烟史的人将会以同样的概率被分到两个子样本中，于是“吸烟史”这样一个干扰因子就被随机化排除了。

随机实验的绝妙之处在于，不单是我们所能想到的，就连那些我们很难考虑到的无数其他干扰因子的影响，也被排除了。

因此，虽然我们可以通过算法针对某些变量找到一些不错的划分，但是这些划分不可能对所有变量都有同样好的效果。这也正是我们需要随机化的原因，因为随机化无论对于我们能考虑到的变量还是没有考虑的变量都有同等的效果。

随机实验在医学研究中也有自己的软肋。根据医学研究的“临床均衡”原则，只有当医学界确实不清楚哪一种治疗方法更好时，随机化分组才是道德上可以接受的。如果研究人员基本确信某药物对某疾病有效，而将一部分人随机化分组到控制组中（也就是说，不给予该药物治疗），这是不符合医疗道德的。

举个例子。为了找出抽烟与心脏病之间的联系，我们不能随机地选出一部分人并建议他们抽烟，因为抽烟有害健康是一个公认的事实。同样，采取随机临床实验研究吸食可卡因与婴儿重量的关系也犯了道德禁忌；研究饮食与死亡率的关系也同样十分棘手，因为这些都直接影响到人们的生命健康安全。

另一个问题是，随机临床实验通常都耗资巨大并且十分烦琐。然而，矛盾的是，如果不做随机临床实验又可能导致错误的研究结论，代价甚至更加昂贵。

当然，有时候不是想做随机实验都可以做的，在很多情况下，随机实验甚至是不能实现的。比如在OK Cupid 的例子中，我们明明知道存在大量的干扰因子，却无法应用随机实验。为什么呢？想象一下，如果系统随机的给女性会员发送赞美她们的邮件，那么OK Cupid可能第二天就倒闭了。

总之，当随机实验的条件满足时，它是解决因果推断问题的黄金法则。然而，随机实验也常常由于道德和现实条件的限制而变得不可行。

平均与个体

随机临床实验衡量的是某一种药物对所有人的平均效用。要想研究关于男性、女性，或者某一年龄段的组别平均效用，则需要采取分组的方法。但即便分组得很仔细，所得到的效用也仍是平均意义上的。换句话说，我们的研究还不能将实验的效用具体到某个人身上。最近一段时间，随伴基因组技术的出现，个性化临床研究开始萌芽并显示出其独特的应用价值。以之前OK Cupid的研究为例，研究的结论是所有男性的平均效果，而不能单独应用在Frank身上。

11.4 A/B 测试

在软件领域，随机实验也称作A/B测试。事实上，我们发现，如果对工程师说“实验”这个词，那意味“尝试新的事物”：如让用户实验同一个产品的不同版本，以推断用户对软件版本的喜爱偏好；而不一定是指一种统计分析方法。A/B测试其实十分易于理解，操作起来也不是很麻烦。事实是，一个简单的A/B实验可以用一个简短的配置文件，外加一个调整参数（如颜色、外观、软件版本等）即可实现。因此，在科技公司内开展A/B测试在某些方面比进行一个临床实验要简单得多。并且，由于和人的生命健康关系不大，因此实验的道德和实际风险都很小。在医学随机临床实验中，我们不能选择让一部分人用药而另一部分人不用；而如果实验的平台是整个互联网，我们可以决定给用户展示什么或者不展示什么，这基本不会引起任何监管的问题。但这些也都不是绝对的，即便是科技公司和互联网公司，在A/B测试时也需要考虑很多问题。

A/B测试理论上很简单，但放到公司层面，实施起来却没有那么容易。公司的产品部门会有很多小组，每个小组否负责产品某方面的特

性。如果这些小组在A/B测试的时候没有协调好，那么A/B测试的效果会大打折扣。比如，用户界面小组总是想测试用户对字体字号的喜好，于是会用到A/B测试。与此同时，内容评分小组想要改善某个推荐算法，也会用到A/B测试，而广告小组则会想方设法提高广告系统的盈利能力，也用到A/B测试。在A/B测试的时候，各个小组关心的结果是一样的：某项改动能否带来更多的用户点击。在测试的过程中，如果某个小组发现，用户点击率确实在某项改动之后有了显著的增加。然而，这个增加的效果到底应该归功于用户界面小组，还是另外两个小组呢？如果小组之间沟通不畅，协调不够，则A/B测试的结果可能会无法溯源。用户界面小组认为是字体的改变带来了更多的用户点击，但实际上很可能是由于推荐算法的改变或者广告系统的升级等多项改变联动引起。

A/B实验的的基本构建有很多方面是要细心考虑的，Diane Tang 等人于2010年所写的论文“Overlapping Experiment Infrastructure: More, Better, Faster Experimentation”（“重复实验构建：更多、更好、更快的实验”）对此有过详细介绍。我们从该论文中摘取一段，与大家分享。

> 摘自“重复实验构建：更多、更好、更快的实验”
>
> 如题，我们进行实验基础构建是为了实现更多、更好、更快的目标。
>
> - 更多
>   实验需具备可扩展性以同时运行更多的实验。然而，灵活性也十分重要，因为不同的实验需要不同的参数设置以及不同的样本量以更好的估计统计显著性。某些实验可能只与程序运行的某个子段有关，而其他实验可能与整段程序有关。
>
> - 更好
>   应该尽量避免明显无效的实验，低效的实验也应该得到尽早地优化或者直接剔除（比如错误代码或者效果明显很差的代码等）。应该用一套标准的程序检验实验效果的好坏，并比较不同实验效果之间的优劣。
>
> - 更快
>   构建实验的方法应该简单易行，以便非工程师

> 不用写代码也能直接上马实验。实验模型的评估应该快准狠。如果是简单的重复性实验，速度是关键。理想的状态下，系统应该不仅能够支持实验，还能够系统的控制项目的预热与爬升。也就是说，实验或者模型的某种变动应该以一种系统性的、容易被理解的方式，逐渐地渗透到项目的其他部分。

一个项目的实验基础架构通常会由一个很大的团队在背后支撑，全天候地做着各种各样的分析工作。虽然是基础架构，却绝非易事。随着社交网络的迅猛发展，实验的基础架构变得越来越复杂。因为网络的先天相关性为实验设计中的独立性假设提出了巨大的挑战。比如说，Facebook设计了一个实验，Rachel被分在了“实验组”，实验处理的内容要求Rachel发表一些特定内容的博客，而Cathy被分在了控制组。实验的随机化原则告诉我们，Rachel和Cathy是相互独立的。但事实上呢？由于社交网络的关联性，Rachel发的博客，Cathy即便是在控制组也难免会看到这些博客，因此她们不是完全独立的：Cathy因为网络的关联性，也接受了一定程度的“处理”。社交网络为很多基础研究都提出了新的课题，有待科学家们进一步研究。

11.5　退一步求其次：关于观察性研究

虽然一般情况下因果关系推断的黄金准则是采用随机实验或A/B测试，但正如我们反复强调的，它们并不总是可行的。有时候我们不得不退而求其次，用观察性研究的方法解决问题。

让我们先介绍它的定义:

> 观察性研究是当控制实验（随机实验）不可行时而采用的一项分析因果关系的实证性研究方法。

许多数据科学研究都是围绕观察性数据展开的。前面所讨论过的A/B测试是个例外。很多时候，你能使用的数据就是你所观察到的数据。很多时候，我们没有能力让时间倒流，或者让同样一件事情重复发生。例如，对于总统大选这样的事件，我们只有观察性的数据，不可能进行任何实验。

众所周知，实验性的数据要优于观察性的数据，因为在实验中，很多因素可以被人为地控制，观察性的数据则不然。然而很多的实验从道德、成本上来考虑是不现实的。因此我们不得不分析手头能够得到

的、观察性的数据。观察性研究就是在这样的一个背景下，为了研究因果关系而退而求其次的研究方法。

即便你可能毫不关心因果推断的问题，而只在意模型的预测效果，预测模型使用的仍然是观察性的数据。观察性研究中可能遇到的问题，在预测模型中同样会遇到。

11.5.1 辛普森悖论

首先，观察性研究中存在着许多陷阱，辛普森悖论就是其中一个。

图11-3是一个简单的散点图，你可以找到一条最佳拟合线来描述是否食用高剂量的某种“不良药剂”会提高犯心脏病的概率。

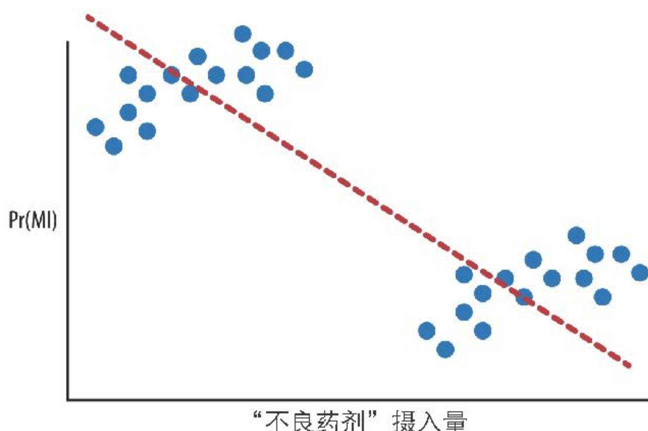


图11-3：患心脏病（简称为MI）的概率与某种“不良药剂”摄入量的关系散点图

从最佳线性拟合的角度来看，上图似乎表明食用剂量越高，犯心脏病的可能性越小。但是，图中的数据形成了两个明显的聚类，如果进一步分析这两个聚类，反而会得出截然相反的结论，这从图11-4可以看出。

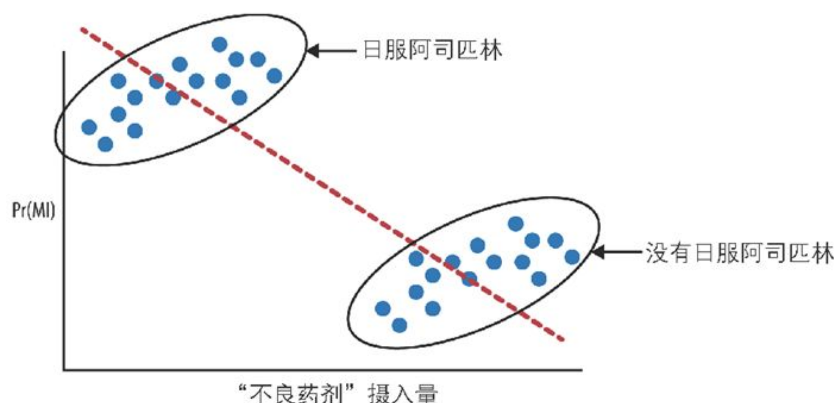


图11-4：患心脏病（简称为MI）的概率与某种“不良药剂”摄入量的关系散点图，此处考虑两个聚类，一类是日服了阿司匹林的患者，一类是没有日服阿司匹林的患者

{60%}

这幅图是有意设计的，因此聚类的问题看起来十分明显。但当数据是多维时，你很难一眼就看出来其中可能存在类似的问题。

在这个例子中，我们可以说“日服阿斯匹林”是一个干扰因子。因此，服用与不服用阿斯匹林的人群不是随机分布的，从图中看来，这甚至导致了因果关系推断的方向都发生了改变。

这里如果你的目的是预测，那么一条如图11-3那样的拟合曲线完全可以满足预测的要求，而且预测效果可能还相当不错。当我们的任务是因果推断时，故事因此发生了根本性的改变，我们不能苟同于那条最佳拟合直线，因为很明显直线的方向与真实因果关系的方向是南辕北辙。

上面的情况在对观察性数据进行回归分析时是极为常见的。如果数据维度很高，你根本不可能知道数据内部到底发生了什么。正如Madigan所形容的，面对高维度的数据，“那简直就是西部荒野，我们对其一无所知”。

情况也有可能是这样，在是否“日服阿司匹林”的两个类别中还有性别的分类，如果再按此分类，你可能又会得到与最佳拟合直线一致的结论：服用剂量越大，犯心脏病概率越小。这种分组因素对因果关系推断的方向性的影响称作辛普森悖论，它在因果推断中有着重要的作用，督促人们考虑可能的、会引起因果关系转向的类别因素。

11.5.2 鲁宾因果关系模型

鲁宾因果关系模型是一个数学模型，在观察性研究中用来确认哪部分信息是可知的，哪部分是未知的。

鲁宾模型可以用来回答这一类的问题："我患癌的原因是因为我曾经有烟瘾。"你确定吗？如果确信，你要能够提供证据支持这个论断。另外一个发问的方式是："如果我曾经不抽烟，我就不会得肺癌。"这是一个很有意思的发问方式，但现有的研究方法还不能回答这样的问题。

定义Z_i为对个体i进行的实验（0 ＝ 控制组，1 ＝ 实验组），$Y_i(1)$为实验组的结果（$Z_i = 1$），$Y_i(0)$为控制组的结果（$Z_i = 0$）。

我们所关心的个体层面上的因果关系（unit level causal effect）就是$Y_i(1) - Y_i(0)$。但是，对于$Y_i(1), Y_i(0)$我们不可能同时观测到Y可能的两个值。

用一个例子来说明：假设我本人为研究个体i，如果抽烟则$Z_i = 1$，否则为0。如果抽烟并患肺癌，则$Y_i(1) = 1$，而如果抽烟但未患肺癌，则$Y_i(1) = 0$。同样的，$Y_i(1)$为1或0，取决于我不抽烟时是否患肺癌。抽烟对患肺癌的总体因果影响是$Y_i(1) - Y_i(0)$。如果确实由于抽烟而患肺癌，它是1；不管抽不抽烟，如果患肺癌（或者不患）都为0；如果抽烟但没患肺癌，则为-1。然而，由于我只可能知道其中一种结果，$Y_i(1) - Y_i(0)$的值无法直接计算得到。

在总体的层面，我们知道有多少人为1。但是从样本的角度来看，我们不能随便将1这个值赋给某人（即我们不可能选择某人，使他/她主动经历抽烟及癌症）。

这个问题也称作"因果关系推断的基本问题"。

11.5.3 因果关系的可视化

我们可以利用因果关系图来呈现因果关系建模的概念。

首先，用W表示所有潜在的干扰因子。这个假设通常是很难站得住脚的，尤其是在流行学的相关研究实例中，潜在干扰因子的个数基本不可能完全确定。关于流行病学研究，会在在下一章做详细介绍。

在关于Frank在线约会的例子中，我们找出了一个潜在的扰乱因子

（他中意的女性本身是否漂亮）。如果考虑得更加深入，我们当然还可以发现更多的干扰因子，如Frank本身是否是个帅哥，或者他最近心情不好，等等。这些因素都会影响到他写邮件的方式（措辞、语调等）以及对方积极回复的可能性。

其次，我们用A表示实验的处理。在这里指是时Frank是否在搭讪邮件中使用了“漂亮”一词。我们通常假设A是二元的（即具有0/1的值）。因此，对于Frank搭讪的女士，如果Frank使用了“漂亮”一词，我们给她赋值1。但这里需要注意的时，即便Frank用“漂亮”形容的是今天的天气，仍给该女士赋值1。也就是说，我们在这里不考虑“漂亮”一次出现的具体语境，只要在邮件中出现，一律赋值为1。

用Y表示实验的“结果”，它同样是一个二元值，只取0/1。对“结果”的定义要十分明确。比如我们定义，在OK Cupid的平台内，如果Frank在发送第一封邮件的一周内收到了该女士的回复，Frank在邮件中向对方索要了电话号码，并且该女士在回复的邮件中提供了自己的电话号码，我们则将1赋给Y。另外还注意，如果一位女士没看到Frank的邮件，但由于其他一些原因还是通过邮件把电话号码告知了Frank，我们仍会给Y赋值1。

因果关系图中的干扰因子，实验的“处理”和实验的“结果”都用节点表示；用箭头表示因果关系的方向。换句话说，箭头的出发节点是因果关系的“因”，而所指向的是“果”。

Frank的例子可以表示为图11-5。

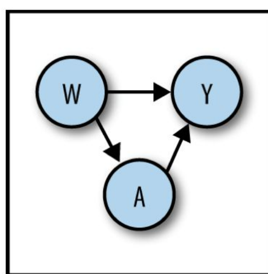


图11-5:一个实验“处理”，一个干扰因子和一个实验“结果”的因果关系图，对应Frank的例子

在OK Cupid的案例中，因果关系图十分简单，只包含一个实验“处理”，一个干扰因子及一个实验“结果”。在实际问题中，因果图会变得非常复杂。

11.5.4　定义：因果关系

假设实验共有100个人，实验的处理是服用“某种药物”，实验的结果是“患癌”与否。实验的最终结果显示共有30人患癌，这意味着患癌率为0.3。因果推断想要回答的问题是：服用该药物是否导致了罹患该癌症？

为了回答这样一个因果推断的问题，从逻辑上来说，我们需要知道，如果没有服用该药物，会有多少人患癌。假设对于同样的100个人的群体，我们规定他们不服用该药物（也许我们是上帝，但即便上帝这么做也是不道德的，人都有生命健康权）。最终的结果是，患癌的总人数减少到20人，患癌率为0.2。我们用两种情形下患癌率的差值表示实验“处理”对实验“结果”的因果影响值。在这里，因果关系的影响值为0.3 - 0.2 ＝ 0.1。



有时候因果关系的影响值也可以用比率值表示，而不一定是差值。

然而，我们并非上帝，不能让同一批人接受药物“处理”的同时还可以用作控制组。现实的情况是，他们要么在实验组，要么在控制组。因此，我们需要一个独立的控制组，在该组内的人们不服用此药物。假设这个控制组中人们患癌的概率为0.1（可以理解为人们自然患癌概率）。那么，通过它，我们可以计算出由于服用该药物而导致的癌症率实际上提高了20%。但这个结论往往是站不住脚的，因为实验组与控制组总有一些不同质的地方我们没有考虑到1。

> 1指的就是干扰因子的影响。

如果实验组和控制组的两批人是完全同质的（这在生物学上基本是不可能的），所有的干扰因子都可以被排除。但是，就像我们说过的，也许上帝可以做到，但是我们做不到。

因此，对于观察性数据来说，最重要的莫过于确定如何把样本分成实验组和控制组。倾向性得分匹配法就是最为著名的方法。从本质上来讲，倾向性得分法是一种伪随机分组法（pseudo-random experiment），人为控制组的分类原则是尽量选择与实验组总体特征

特别相像的个体（这些个体以同样的概率出现在实验组和控制组中）。具体该怎么做呢？注意前面提到的“特别相像”的概念，有很多方法可以找到相像的个体，逻辑回归是其中较为常见的方法。

倾向性得分匹配法分为两个步骤。第一步是用逻辑回归计算每个人接受实验“处理”的概率；然后我们将接受实验与未接受实验的人匹配起来。其实，被分在“伪控制组”的个体具有同等的可能性被分在“伪实验组”，只是实际上没有而已。第二步就是假设样本已经被分成了实验组与控制组，可以应用常规随机实验的分析方法，分析分组之后的数据。

例如，如果我们想要研究抽烟对患肺癌的影响，由于观察性研究的限制，我们需要尽量找出（或者观察到）有同样抽烟可能性的人。因此，能够搜集到的个人信息越多越好：包括性别、年龄、父母是否抽烟、配偶是否抽烟、体重、饮食习惯、运动习惯、一周工作的小时数、血液检测结果等。第一步的逻辑回归模型，Y变量是个体是否抽烟。逻辑回归模型可以根据个体的体征输出个体抽烟的概率值。这些概率值就是每个个体的倾向性得分，得分值的作用是为了匹配样本，以尽量保证分组的同质性。这里我们假设所有的个体特征变量都是可以观测到的，但事实上，对某些变量的观测和样本收集会比较困难。

这也是倾向性得分匹配的先天性缺陷：我们永远无法确信已经考虑到了所有该考虑的因素。然而，它的优越性在于，如果可能的干扰因子都能被考虑到，那么在倾向性评分匹配模型背景下的因果关系推断是合理和有效的。

复杂的数据和复杂的因果推断问题，也对应复杂的倾向性得分匹配方案。一些关于倾向性得分模型的程序包也设计得很好，很多匹配都可以自动化地完成，使用者不需要关注太多细节。当然，使用何种模型计算倾向性得分，以及该模型中的Y变量（要与相应的因果推断对应）还是需要使用者自行设定的，除此之外的事情都很具体化。

在之前的约会例子中，我们需要用什么样的数据来估计因果关系呢？一种办法是用类似土耳其机器人这样的工具，人工浏览所有收到Frank邮件的女士的资料，并且将长得漂亮的标记出来。这样我们就可以直接推断收件人本身是否“漂亮”这个扰乱因子对分析结果的影响。这在因果推断研究中叫作分层法，下一章会详细论及。该方法虽然有效，但是也会带来不少问题。

11.6　三个小建议

关于建模，Ori与大家分享了三个小建议。

第一， 当进行因果推断时，深入地了解数据的生成过程至关重要。因为任何模型都会有相应的模型假设，数据本身的数据生成过程可能会明显背离这些假设。如果假设明显不符合数据的生成模型，那模型的使用就应该打上问号。

第二， 数据分析的第一步应该置身数据之外，弄清楚到底想要分析的问题是什么。可以把问题写下来，这会帮助你思考，然后再一步一步地思考使用什么样的工具解决这些问题。在使用工具分析数据的过程中，要不时地回头想想当初想要回答的问题，以及正在做的事情是不是在正确的轨道上。这听起来很有道理，也稀松平常，但人们往往都会忘了这么做，忘记了自己分析问题的初衷。

最后，当你运用算法分析数据时，不要被算法和代码冲昏了头脑。不要以为只要算法收敛，模型参数估计显著就一切大吉了。在数据分析时，要时刻保持一颗清醒的头脑，人脑可以发现电脑所不能发现的逻辑性的、常识性的错误；人也应该在数据分析中扮演主导型的角色。

第 12 章　流行病学

本章的贡献者是David Madigan教授，他是哥伦比亚大学统计学院院长。Madigan的研究领域包括贝叶斯统计、文本分析、蒙特卡洛模拟方法、药物警戒系统、概率图模型等。在这些领域，他发表了不下于100篇学术文章。

12.1　Madigan的学术背景

Madigan 1980年毕业于都柏林三一学院（Trinity College Dublin）。他本科主修数学，但在最后一学年选修了一些统计学的课程，并且自学了一些关于计算机的知识，包括Pascal语言、操作系统、编译器、人工智能、数据库理论等。大学毕业后，Madigan先后就职于一家保险公司和一家软件公司，前前后后干了6年。期间他的主要工作是对专家系统（expert system）进行研究。

那个时候，个人计算机还没有问世，编程都是通过脚本语言在大型计算机上实现的。Madigan在保险公司工作的时候，主要从事保险产品定价策略方面的编程工作。另外，他之前还做过一个污水处理系统的项目，并且学了一些可视化方面的知识。他知道如何利用计算机上的显卡进行编程，但是对于数据，他那个时候还接触得很少。

工作了几年之后，他回到了都柏林三一学院完成了博士学位。博士毕业之后，他选择进入学术界，并且在华盛顿大学获得了终身教授的职位。那个时候，机器学习和数据挖掘方兴未艾，他很快就对这两个研究方向产生了强烈的兴趣。他之前还担任过KDD数据竞赛的大会主席。此间他学会了使用C、Java、R和S+编程。但是即便是当了教授之后，他还是很少和实际的数据打交道。

他说，在刚开始他是一个典型的学者：知道如何编程，却在接触到一个大型的医疗数据时不知道从何做起。因为当时这个医疗数据包括50个从不同数据库中收集的不同格式的数据，面对这样的数据Madigan当时一筹莫展。

在2000年的时候，他去AT&T实验室工作了一段时间。据他描述，AT&T实验室的研究环境是完全学术化的，他在那里学习了Perl语言、awk语言和一些Unix的基本知识。他甚至还做了很多网络检索方面的工作。

在那之后，他选择了自己创业——和朋友一起成立了一家互联网公司。该公司的主要产品是一款消费者活动实时可视化系统。

创业的经历给了Madigan很多分析大型医疗数据的机会。他曾在不少医疗纠纷的审讯中出庭作证，向法官提供医疗实验数据的分析结果。他说，做这些工作让他在数据解释的问题上大开眼界："把逻辑回归解释给法官听，比我在这里给你们讲课要还要难上加难。那是一种完全不同的、全新的挑战。"Madigan觉得，简洁明了的可视化对解释分析结果确实有很大帮助。

12.2　思维实验

假设我们手头有一套详细的医疗诊断数据。这套数据是关于每个人的医疗历史的详细记录（统计学中称作纵向数据）。样本量大约为8000万个病人，数据记录的内容包括每位病人的处方药单、每次看病的检查结果、每一次医院或者医生家访的检查结果、手术的数据等。每条记录发生的时间都有详细的记录。面对这样庞大的数据，我们能干些什么呢？

从现实情况来看，我们还在延续着中世纪以来的传统，医生对这些数据充耳不闻，看病只依赖于自己的主观判断。但是我们是否能够做得更好一些呢？这些数据能否带来更好地医疗保健体验呢？

这其实是一个非常重要的全民医疗问题。而对于医疗保险公司来说，充分合理地利用好这样的数据至关重要。一个感兴趣的方向是，如果在需要住院之前，医生能够及时地介入病人的治疗，这必将节省大量的医疗资源。要实现这个目标，是需要大数据做分析支撑的。

Kaggle之前有一个关于医疗数据的分析竞赛，竞赛的题目叫作"Improve Healthcare, Win $3,000,000"（"改善医疗保健，赢得300万美元"）。该竞赛的题目要求参赛者能够准确地预测人们在第二年是否需要看病。当然，由于隐私问题，实际竞赛数据中关于个人隐私的数据都被舍去了。

涉及公民个人隐私的数据如果被非法使用，会产生相当严重的后果。比如说，不法分子可以从数据中找到那些有钱的病人，并想方设法敲诈他们。不良的保险公司可能会利用这个数据找到那些医疗风险大的人，取消与他们的保险合约。这些既是法律问题，又是道德问题。在涉及大量个人隐私信息的数据的使用上，我们既要恪守道德标准，也不能触犯法律的红线。

12.3 统计学在现代

想象一下，就在20年前，统计学研究在学术界是一番怎样的光景？统计学家要么坐在办公室里证明各种各样的定理，要么在构思某种新的假设检验的方法，或者是缺失值处理的方法。他们根本不会、也不需要跟数据打交道。学术界的统计学与实际的数据分析是脱节的，统计学也不需要领域专家的帮助，统计学家就只沉浸在自己的小世界里。

现在的统计学研究已经发生了本质的变化。顶级的统计学杂志开始看重统计在实际问题上的应用。如果论文是社会科学家（或者很多其他应用科学家们）与统计学家一起合作的成果，往往都很受欢迎。Madigan也指出，在医学研究领域，统计学家正在扮演越来越重要的角色。

Madigan觉得现在的机器学习在学术界的发展情况十分类似于20年前的统计学。机器学习是一个新的学术研究领域，学术会议的发展也已经比较成熟，但是作为机器学习界的一员，Madigan还是觉得它仍未脱离当初统计学家们闭门造车的风格：大家都在绞尽脑汁开发新的算法，并千方百计地找数据验证这些算法。Madigan觉得，机器学习的发展如果不与其他应用领域结合起来，很难有长足的进步。

Madigan指出，现在很多的统计学家都不注重提高自己解决实际问题的能力，然而现实世界的发展已经对统计学家提出了更为实际的新要求。（当然，Madigan的同事Mark Hansen是一个明显的反例）。

12.4 医学文献与观察性研究

观察性研究的内容我们在上一章已经讨论过，它是统计学中的一个十分重要的研究方法，也是医学研究的标准方法之一。观察性研究的结果对于医学培训、临床以及政府的医药智力起着举足轻重的作用。

比如说，Jane Green等人有一篇合著的论文，题为“口服磷酸双酯提高患食管癌、胃癌和大肠直肠癌的风险：基于英国初级医保数据的病例对照分析研究”（参见http://1.usa.gov/16UfNjZ）。Madigan看完这篇论文后总结说：这篇论文处理的问题与当初对阿司匹林的研究一样，就是干扰因子的问题。该论文的结论是，口服双膦酸盐显著地提高了患这些癌症的风险，提高的幅度起码达到了10%。

这篇文章刊登在《纽约时报》某期的首页，研究者是几个没有利益冲突的学者1，研究的样本来自数以百万计的患者服药数据。但即便是

这样，其研究结论也很可能是错误的，并且之后的研究成果也很可能推翻这一结论。

> 1没有利益冲突代表作者互相之间没有利益瓜葛，既没有侵权也没有利益瓜分之嫌，因此研究的结果较为可信。

这样的例子还有很多很多。这其中是一个没有引起人们足够重视，却又十分重要的问题。

政府每年在医药研究上都会花去大笔的金钱，因为医药研究的结论会关系着无数人的生命健康。因此，我们必须在正确的理论指导下，做严肃的医药研究。

12.5 分层法不解决干扰因子的问题

流行病学研究是一个与干扰因子做斗争的过程。但是可惜的是，现有的流行病统计学的研究方法还不能很好地解决这个问题。其中最常使用的方法是分层法。举个例子来说，如果我们觉得性别是可能的干扰因子，在研究的时候就按照性别把样本分成两层，并在分析的时候根据不同性别样本的权重调整估计值的结果。

然而，当实验的结果中数值比较小，或者不同层总体之间差异显著时，分层法会影响因果分析的有效性。

表12-1是某实验结果的汇总表，我们用这个表举一个例子。

表12-1：某实验汇总表

		实验组：服药 / 对照组：未服药		
Y = 1 / 30				
Y = 0 / 70				
P(Y = 1) / 0.3 / 0.2				

从表中可以得出，该实验的实验组和对照组的人数均为100人，表中间的两列实际上是不可观测的。由汇总表的结果可以得到，因果关系效果值为0.3 - 0.2 ＝ 0.1，也就是10%。

然而，当我们按照性别将汇总表成两个子表之后，问题就来了，尤其是当这些表中出现很多很小的数目的时候。分层之后的男性子表见表12-2，女性子表见表12-3。

表**12-2**：分层结果：男性子表

	[illegible]组：[illegible]服药	
[illegible]= 1		
[illegible]= 0		
[illegible](Y = 1)		

表**12-3**：分层结果：女性子表

	[illegible]组：[illegible]服药	
[illegible]= 1		
[illegible]= 0		
[illegible](Y = 1)		

由表12-2得知，男性的因果关系效果值为0.3 - 0.25 ＝ 0.05，而表12-3告诉我们相应女性的因果关系效果值为0.3 - 0.187 5 ＝ 0.112 5。最后的研究报告可能会宣称该药物对女性的效果是男性的两倍多。

换句话说，分层的想法有时候不仅没有解决问题，反而带来了很多新的问题：因果关系的估计结果往往变得扑朔迷离。因此，当你在解决干扰因子问题时，至于到底该不该用分层法，需三思而后行。

人们在实证中到底如何处理干扰因子的问题

虽然说分层法存在一些潜在的问题，但是在实际分析中，分层法仍然是使用最多、应用最广泛的方法。对于明显的或潜在的干扰因子，都可以针对该因子变量将样本分层再做分析，或者针对该因子做一些模型层面的调整（比如上一章详细讨论过的“倾向评分匹配法”）。因此，如果我们觉得某实验中，服用阿司匹林是一个干扰因子，那么可以针对该因子将样本分层以排除该因子对因果关系推断的影响。

这里有一个有趣的例子（http://goo.gl/3VgRi0）：某项实验的目的是研究口服避孕药与静脉血栓的关系。研究人员在实验中考虑了一些他们认为可能的干扰因子，并得出了以下结论：

> 在调整了人们服药长度的可能影响之后，实验的结果显示口服避孕药的女性患静脉血栓的几率是不服用口服避孕药女性的两倍。

该研究的结果曾经轰动一时，美国广播公司对此也有过报道。但是，该实验的研究人员在考虑干扰因子的时候显得有些不走寻常路。按理说，是否服用其他药物，比如阿司匹林这样的常用药，应该是一个明显的干扰因子。因此，即便分层法可以在一定程度上解决干扰因子的问题，但是应该如何选取干扰因子却是一个令人头疼的问题。另外一项研究中，研究人员将实验对象的静脉血栓史作为干扰因子，得到了完全不同的结果。

这个例子告诉我们，干扰因子的选择对实验结果有着直接影响，选择不同的干扰因子，得出的结果可能有天壤之别。譬如之前的口服磷酸双酯的研究课题，研究人员选择了吸烟与否、饮酒与否以及BMI这样的干扰因子。但是如果另外一个研究选取另一批干扰因子，得到的结果可能大相径庭。从这一点来看，处理干扰因子是一个无解的问题，我们几乎不可能把所有的干扰因子都考虑进来。这是一个哲学问题，而不是一个科学问题。

Madigan和几位合作者曾经做过一项研究，他们让一批流行病学家跟对同一组课题设计5个实验。最后发现，这些设计在底层环节上都非常一致，但是对于考虑什么样的干扰因子，每个人都自诩专家，并宣称自己的设计是最合理的。

12.6　就没有更好的办法吗

我们无法直接回答这个问题，但是可以给你一个很醒脑的例子。Madigan和他的合著者做过另外一项研究，他们选取了50个有关药物及其副作用的课题（比如抗生素与消化道内出血的关系等）。每个课题都放在9个数据集上做接近5000个不同的验证分析。

比如，血管张力素转化酶抑制剂会引起心悸，针对此课题他们在9个数据库上做了接近5000个验证分析。最小的数据库有400万个病人的数据，而最大的一个数据库样本量达到了800万人。

针对这一个例子，某一个数据库的分析结果显示血管张力素转化酶抑制剂引起心悸的几率是对照组的3倍，而另一个数据库的分析结果显示是6倍。这里虽然研究的结果在不同数据库上有所不同，但起码它们的指向都十分一致，那就是该药物确实增加了心悸发生的几率。

然而，对于50个课题中的20个课题，在不同数据库上得到分析结果似乎完全是随机的。有些具有统计显著性，有些没有。有些具有正的显著性，有些具有负的显著性。也就是说，你能想到的结果都能在某个

数据库的某项验证分析结果里找到对应的。图12-1展示了这50个课题的统计显著性分布。刚才关于心悸的课题位于该图的最上方。

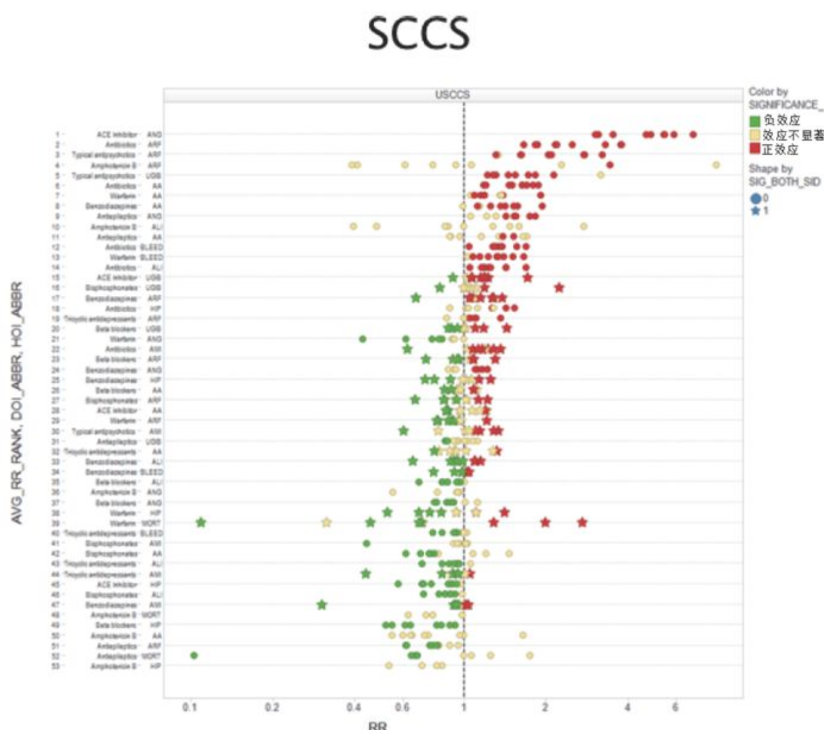
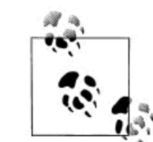


图12-1：50个课题的统计显著性分布图



数据库的选取对研究的结果有直接影响，但是现在的大多数论文都对此条避而不谈。

Madigan团队接下的研究更加有说服力。他们详细地考虑了每个实验的性质和研究的可能性，只要有模棱两可的问题，那么所有的可能性都会考虑进来。具体说来，他们考虑了这样一些可能性：使用哪个数据库；考虑哪些干扰因子；检查的时间窗长度（比如一个心脏病人在停药后一周发病的概率和一个月发病的概率明显会有所差异）等。

他们得到的结论是：所有的实验结果都可能有两个对立的解释！

回到刚才口服磷酸双酯的例子。一团队的论文（http://1.usa.gov/16UfNjZ）宣称口服磷酸双酯会增加患癌的风险，而另外一篇发表于JAMA的论文（http://1.usa.gov/1hi2kbj）则表示口服磷酸双酯不会增加患食管癌的风险。他们分析的数据甚至来自于同一个数据库，却得到了截然相反的结论。其实这样的研究成果具有对立性的现象在医学界还有很多。

12.7 研究性实验（OMOP）

为了直接解决上面所提的这些问题，或者说起码在一定程度上突破现有研究理论和方法的瓶颈，Madigan加入了一个叫作OMOP（http://omop.org/）的研究项目并担任了首席研究专家。他的使命是开发和实现一些可以用作观测数据分析的新统计方法。这套统计方法是OMOP项目的核心贡献。OMOP的全称是“Observational Medical Outcomes Partnership”，我们下面稍作介绍。

> **关于OMOP**
>
> 2007年的时候，由于认识到电子健康信息（EHR）数据和其他大型的公平健康数据规模正井喷式地增长，这样的数据为医药和人体健康方面的研究提供了宝贵的契机，国会决定以FDA牵头，打造一个新的医药监控项目,以更好地保护公民的健康安全。FDA随后牵头了一些项目，包括非常著名的Sentinerl项目。该项目旨在搭建一个全国性的数据网络。
>
> 在该背景下，国家健康委员为与PhRMA以及FDA发动了OMOP项目。该项目是典型的公私合作模式。OMOP项目已经为业界带来了一些令人瞩目的研究成果，其中的一项研究成果确认了研究和分析超大型同质性数据的流程和方法。
>
> OMOP项目的贡献者来自于流行病学、统计学、计算科学以及许多别的学科。他们通力合作，只为了回答下面这些问题：医药研究者能从这些大型健康数据中得到哪些有用的信息？有没有一个普适的流程和方法可以用来研究和分析不同数据库的数据？研究的结果能不能被很好地验证？
>
> 如果对上述问题的回答都是肯定的，那么这将为医药

和健康研究领域带来翻天覆地的变化。从长远来看，整个国家的医疗系统以及医疗监管系统也将发生质的变化，人们的健康安全将得到更好的保障。

Madigan和他得团队选取了10个大型医疗数据库，这些数据库来自于保险公司和EHR等机构，涵盖了大约两亿人的资料。这绝对是一个大数据！

他们详细地研究了观察性研究中经常使用的模型，并用上面的大数据进行了一一验证。最后他们选取了14个常用的纵向数据流行病学模型。模型的细节被全部自动化，并且每个模型都有大约5000个可调参数。

这样做是为了验证已有模型的预测效果是否准确。

因为公认的研究成果可以用来验证这些模型的效果。他们选取了10个被广泛研究的药物，包括血管素转换酶抑制剂、华法令阻滞剂以及苄丙酮香豆素等；以及10个用药症状，包括肾功能衰竭、住院和出血症状等。

对于这10类药物来说，其中某些的副作用是已知的。比如说，华法令阻滞剂会导致血管壁变薄以及失血。类似这样的已知的副作用还有多达9个。

另外有44个已知的无副作用药物，这些药物我们有充分的证据相信他们不会对人体产生副作用，最起码不会产生上述10种用药症状。

验证的方法很简单：在所有10个数据库数据上运行5000种常见的流行病学分析，并验证每个数据库上每种分析的预测效果。这有点类似于第4章讨论过的垃圾邮件过滤模型的测试问题，数据被分为训练数据集和测试数据集。模型的估计只用到训练数据集，而测试则是在不同于训练数据集的独立的测试数据集上完成。

每次测试都只输出两个统计量：相对风险值（Relarive Risk，RR）1和预测误差。

> 1相对风险值是因果关系的测度指标

可以看出，这个项目的最终目的是验证已有流行病学模型的实际预测效果，这有点类似于John Ioannidis的工作（http://stanford.io/15LfJDL）。



为什么这个项目到现在才做？

这涉及利益冲突的问题。没有人愿意去验证他们的模型是无效的，他们自然也不会提供数据给想要验证的人。而且从规模上来看，这个项目是巨大的，其耗资近2500万美元。然而，这笔钱看起来似乎很多，但是与那些已经完成的（结果可能是错误的）研究的经费比起来，这简直就是九牛一毛。

这个项目购买了10个数据库中的所有数据，所有的模型都被自动化，他们还使用了亚马逊的云计算服务用来加快模型的运行速度。他们甚至开源了该项目所有的原始代码。在项目的第二阶段，他们把研究对象缩减到了4个用药症状，并且绘制了所有模型汇总的ROC曲线（这意味着这条曲线价值2500万美元！），见图12-2。

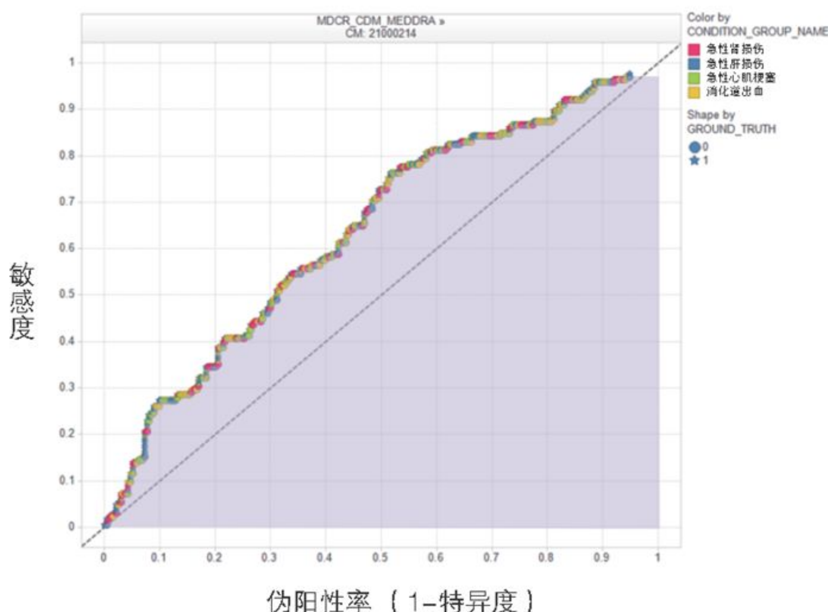


图12-2：价值2500万美元的ROC曲线图

为了理解这幅价值连城的ROC图，我们首先需要定义一个阈值，比如说2。也就是说，如果模型的相对风险值大于2，则模型所估计的因果

关系可以解释为“恶化作用”，如果小于2，则为“促进作用”。阈值的选取对模型结果的解释有着直接的影响。

如果阈值设定为10，那么没有任何一个模型的相对风险值可以达到10，因此所有的模型结果都是有“促进效果”。另外需要注意的是，由于一些药物已经退市或者停产，他们的市场保有量为0或者极少，因此模型的“敏感性”自然很低，因此模型很难发现有任何效果。

当然，低“敏感性”也意味着伪阳性的值也很小。

同样的道理，如果阈值设定为-10，那么所有模型的相对风险值都大于-10，所有模型得到的都是“恶化作用”。此时虽然模型的敏感性”达到了100%，但是伪阳性的值却变大了。

因此，阈值的设定相当于在模型的“敏感性”和伪阳性中做出权衡，这就是ROC曲线的核心内容。我们可以找到一个最佳的阈值1.8，其“敏感性”为50%，伪阳性为30%。



如果你是FDA，这样的ROC曲线是不能过关的。因为30%的伪阳性不在FDA的考虑范围之内，会被立即否决。

在前面的章节我们讨论过，相对于ROC本身，可以用AUC来评价模型的预测效果。AUC为1的模型是最佳模型，如果模型的预测效果十分平庸，则AUC接近于0.5。0.5的AUC代表该模型与瞎猜没有任何区别。

上图的AUC值为0.64。在研究团队（包括David Madigan）运行过的5000个分析中，0.64其实是其中效果最好的模型。

但是需要注意的是，0.64的AUC是建立在对每个数据都应用同一个模型的情况。可以想象，现实中使用的很多模型也基本与瞎猜没有任何区别。

然而，没有任何一个流行病学家会这么干。对于手中的数据，他们总是想找到一个最好的模型以便得到最好的预测效果。他们通常都能得到更好的结果，以上面医疗保健的数据为例，在预测急性肾损伤时，

他们使用的模型的AUC达到了0.92（见图12-3），模型的“敏感性”为80%，伪阳性值为10%。

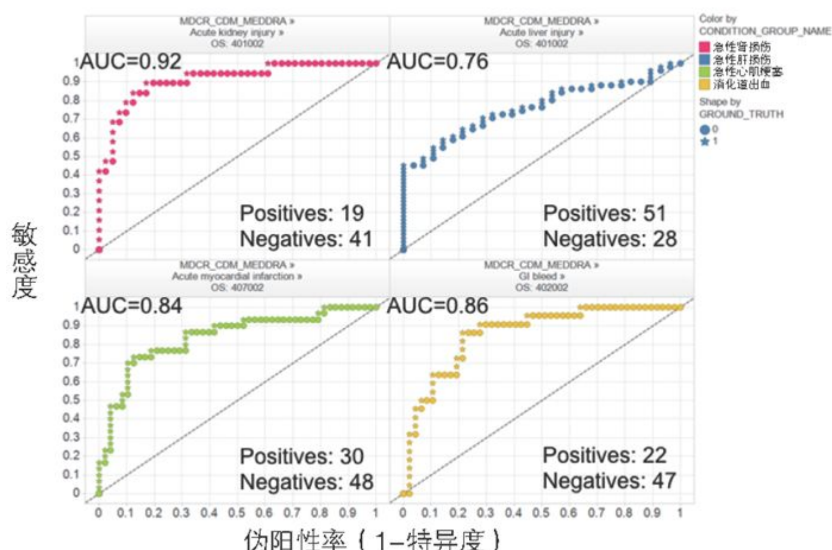


图12-3：针对性的选取模型后，模型的预测效果可以得到较大改善

模型的验证使用了交叉验证法。最终的最优模型是一个叫作“OS”的方法，该方法在给定病人的病历后，在病历期内进行比较（因此，病人在服药阶段和不服药阶段会有较大差距）。不过该方法还没有引起大家的注意。

有意思的是，大多数流行病学家根本不承认这项研究的结果！

如果感兴趣，你可以去该项目的官方网站（http://elmo.omop.org）看看，上面有每个数据以及每个模型的AUC曲线。模型使用的数据大致截止于2013年中。如果有最新的数据，更新所有的分析，结果可能会有所不同。

12.8　最后的思维实验

在上面的OMOP项目中，每个数据库都运行了近5000种分析。有没有可能将这些分析捆绑起来做呢？或者说，用一种类似模型投票的方式，赋予模型不同的权重以达到更好的分析效果。因为该项目的源代码是开源的，说不定你可以就这个想一想题目写一篇博士论文呢。

第 13 章　从竞赛中学到的：数据泄漏和模型评价

本章由Claudia Perlich贡献，过去几年，她一直担任M6D公司的首席科学家。在这之前，她供职于IBM某研发中心的数据科学小组，在电视节目*Jeopardy!*上击败人类赢得比赛的Waston系统就诞生于这个研发中心，但是Claudia并没有参与那个项目。Claudia拥有计算机科学硕士学位，并在纽约大学获得信息系统博士学位。现在她在商学院开设了一门数据科学课程，课程的重点是如何评估数据科学工作的价值和如何管理数据科学家。

Claudia蜚声数据科学界，因她在数据挖掘竞赛中是常胜将军。她曾经赢得过2003年、2007年、2008年和2009年的KDD Cup竞赛，还赢得过2005年的ILP Challenge、2008年的INFORMS Challenge和2010年的Kaggle HIV比赛。

近些年Claudia转而成为这些赛事的组织者。她先是作为组织者参与了2009年的INFORMS Challenge，继而又参与了2011年的Heritage Health Prize。最近Claudia宣布不再参与数据挖掘竞赛，但我们的课程有幸请她来，跟大家分享一些经验见解，看看我们能从数据竞赛中学到什么。在数次竞赛中，Claudia获益良多，特别是数据泄漏和如何评价她在竞赛中提出的模型。

13.1　Claudia作为数据科学家的知识结构

Claudia先询问了其他数据科学家的参考点，根据其知识结构来评估他们在数据科学界所处的位置。（她的知识结构如表13-1所示。对比了第1章讲到的数据科学家的知识结构，Claudia说："有一项最重要、最难形容的技能没有出现在你们的知识结构中，那就是数据。"）她认识一些世界顶级的数学家、机器学习专家和统计学家等。在知识结构表各项技能的描述中，她是以专家的标准来要求自己，还是以她所处领域的平均水平，或者只是以一个普通人作为参考？

表13-1：Claudia的数据科学知识结构

		[illegible]		

可视化	我会做可视化，但是我不相信可视化
计算机科学	我有计算机科学专业的硕士学位，我也可以随手编写一些代码，但不是那种产品级的代码
数学	数学是很早以前的事了
统计学	我没接受过正规训练，很多东西都是边做边学，有时候也要靠自己很好的直觉
机器学习	
[illegible]	你确定这个问题没问错吗
讲演	
数据	

13.1.1 首席数据科学家的生活

过去，Claudia花了很多时间和精力在预测建模上，包括参加一些数据挖掘比赛，为一些学术期刊、学术会议（比如KDD）撰写论文，做演讲，写专利，在大学教书等。但她最喜欢干的事是挖掘数据的含义，她喜欢直接面对数据，经由数据了解世界。

Claudia有15年的数据处理经验，期间她通过深入研究数据的生成过程，培养出一种对数据的直觉。数据生成过程是整个研究问题中的关键一环。她花了大量时间来思考模型评价过程，从而也培养出了对模型的直觉。

Claudia的主要技能包括使用Unix、sed、awk、Perl和SQL对数据进行处理。她能使用各种方法进行建模，包括逻辑回归、*k*最小近邻算法等，最重要的是她花了很多时间将所有的事情都串起来。整理数据、为数据进行建模大约花去她40%的时间，这时她将自己称作“贡献者”。她还要花40%的时间写文章、做演讲，大多数时候是代表M6D公司和外界进行交流，这时她称自己为“大使”。还有20%的时间用来与她的数据科学团队一起工作，这时她的角色则是“领导”。

13.1.2 作为一名女数据科学家

女性在数据科学领域同样可以工作得很出色，她们的直觉在这里是有用的，并且常常被用到。当事情不对劲的时候，她们可以嗅出空气中异样的气息，当然这和使用算法严格来进行证明是两回事。通常，人们更容易记住女性，即使该女性不一定能记住他。Claudia愉快地承认了这个事实。但是，她之所以有现在这样的成就，根本原因还是她很优秀。

在学术界，Claudia曾发表过不少文章，因此熟知在期刊之类的地方发文章的流程。她讨论了在学术期刊或者学术会议上发文章时，两性

是否平等。在过去，这是一个双盲的过程，但是现在更多是单向的。而且根据Shawndra Hill和Foster Provost在2003年发表的一篇文章，根据文章的引用文献就能猜测出文章的作者，而且准确度达到40%，如果该作者发表过多篇文章，预测准确率还会更高。希望这个方法不会在审阅文章时被用到，这只是为了说明，盲审不一定能解决问题。最近，文章开始允许署名，希望这一举动不会造成过多的偏见。Claudia承认自己在一些研究机构中曾遭遇过少许偏见——根据她的经验，某些研究机构的准备工作做得更好。

13.2　数据挖掘竞赛

Claudia对于不同类型的数据挖掘竞赛做了总结。第一种类型是“无菌型”的。数据是事先准备好的、干净的，对错误的衡量也有统一的标准，特征变量常常是匿名的，这是一个纯粹的机器学习问题。

这种类型竞赛的典型例子是KDD Cup 2009和Netflix Prize，还有一些Kaggle竞赛。在这种比赛中，重点是算法和计算。获胜者通常组合了一堆复杂的模型，调用了庞大的计算资源。

KDD杯数据分析竞赛

历届KDD杯数据分析竞赛的题目和对应的数据集都可以从以下网站获得：http://www.kdd.org/kddcup/index.php。下面是历届比赛的题目：

- KDD Cup 2010：Student performance evaluation
- KDD Cup 2009：Customer relationship prediction
- KDD Cup 2008：Breast cancer
- KDD Cup 2007：Consumer recommendations
- KDD Cup 2006：Pulmonary embolisms detection from image data
- KDD Cup 2005：Internet user search query categorization
- KDD Cup 2004：Particle physics; plus protein homology prediction
- KDD Cup 2003：Network mining and usage log analysis
- KDD Cup 2002：BioMed document; plus

gene role classification
- KDD Cup 2001：Molecular bioactivity; plus protein locale prediction
- KDD Cup 2000：Online retailer website clickstream analysis
- KDD Cup 1999：Computer network intrusion detection
- KDD Cup 1998：Direct marketing for profit optimization
- KDD Cup 1997：Direct marketing for lift curve optimization

与之相反，另一种是“现实世界”中的数据挖掘竞赛。面对的是原始数据（数据经常分散在不同的表中，而且合并起来很困难），需要自己建立模型，根据任务特性评价模型的好坏。这种类型的比赛更好地模拟了现实世界，这就回到了Rachel在本书前面的一个思维实验：如何在课堂上模拟一个数据科学家所要面对的混乱局面。这需要在长期处理混乱情况的过程中不断历练。

这种类型的例子有2007年、2008年和2010年的KDD Cup比赛。如果你参加的是这种类型的比赛，那么就需要了解问题所处的领域，分析数据，建立模型。获胜者会是那些最懂得如何根据实际问题对模型进行调整的人。

Claudia更喜欢第二种类型的竞赛，因为这和我们的生活贴得更近。

13.3　如何成为出色的建模者

Claudia认为，数据和领域知识是数据科学家所需具备的最重要的技能，但是这些技能是老师教不来的，只能通过不断积累得来。

Claudia通过参加各种数据挖掘竞赛学到了很多东西，而这些东西在学术界常常被忽视。

- 数据泄漏
  数据泄漏是参赛者最好的朋友，也是组织者和出题者的噩梦。数据总是或多或少有些问题，Claudia将发现这些问题变成了一种艺术，她总能指出准备竞赛的人在处理数据时的懈怠和马虎。

- 真实世界中评价模型的标准
  标准的模型评价标准有均方误差（MSE）、错分率和曲线下面积（AUC）等，在真实世界中对模型做评价，有时候要越过它们，选择其他标准。比如，利润可能是现实生活中一个更有效的评价标准。

- 特征变量创建和变换
  真实数据很少那么规整（看起来很漂亮的数据），如何解决这个问题现在还是个挑战。

13.4　数据泄漏

在2011年的KDD大会上，Claudia和Shachar Kaufman、Saharon Rosset共同发表了一篇文章："Leakage in Data Mining: Formulation, Detection, and Avoidance"（"数据挖据中的数据泄漏：公式化、检测和避免"）。他们提到了另外一个作者Dorian Pyle，他写过很多关于数据挖掘中数据准备阶段的文章。在这个过程中他发现了一个现象，即很多事情变得不合时宜，他将这称为"时空错乱"。他说很多数据好得难以置信，但其实这种数据的存在本身就是一种漏洞。Claudia和另外两位作者将预测建模中的类似现象称为"数据泄漏"。Pyle建议借助探索性数据分析找到数据泄漏的源头，而Claudia他们倾向找出对付数据泄漏的方法。

数据泄漏是借助数据或信息做出预测，虽然结果正确，但其实你能够获得这些数据并据此进行预测，这本身就是不合适或者行不通的。不光在比赛中，在真实环境中，这也是建模时的一个大问题。数据泄露通常是因果颠倒的产物，让我们通过几个例子，看看这是怎么发生的。

13.4.1　市场预测

有一场比赛是预测标准普尔指数（S&P）的涨跌的，获胜者的曲线下面积（AUC）达到了惊人的0.999。股票市场几乎是随机的，出现这种情况要么是因为有人很有钱，要么是因为什么地方出现了错误（提示：那次是有地方出错了）。

在过去一段"美好时期"，参赛者只要找到泄露的数据就能赢得比赛。在这个案例中，虽然我们不清楚到底是哪里产生了数据泄漏，但是通过持续不断地对数据进行分析，有可能就会发现数据集中的有些信息可以对预测标准普尔指数产生决定性作用，但是在真实情况下做预测

是无法拿到这些信息的。举这个例子的用意是说明，比赛中有人取得如此高的AUC一定是依赖于某些数据泄漏，在真实环境下如此建模是行不通的。

13.4.2　亚马逊案例学习：出手阔绰的顾客

这个比赛的目的是通过历史购物记录，预测出哪些顾客会在亚马逊网站上花更多的钱。数据由不同商品种类的交易记录构成。其中赢得比赛的一个模型将“Free Shipping ＝ True”认定成一个关键的预测变量。请注意，只有购物金额在一定数额之上，比如说50美元，才能享受免费送货。

哪里不对劲呢？关键是免费送货是因为花了很多钱的结果。不能因为这样一种相关性去建立模型，比如，这种模型对新顾客就不适用。这里要注意的是，时间戳的作用并不大。包含“Free Shipping ＝ True”的记录和购买行为是同时发生的，用这样的记录做预测是没有意义的。你需要使用以前的数据去预测未来。困难在于收集到的数据中混入了包含免费送货的记录，这些记录必须被人工剔除，而作为模型的建立者，需要深思熟虑，了解你的数据。如果没有把数据泄漏的情况考虑在内，很可能将免费送货作为一个变量来建立模型，并且发现它的预测效果很好。但是，当在生产环境中使用该模型时，你无法知道购物者是否会得到免费送货的待遇。

13.4.3　珠宝抽样问题

还是一个在线零售商的例子。这次是预测哪些顾客会购买珠宝。数据依然由各类商品的交易记录构成。其中一个模型在sum(revenue) ＝ 0时，预测结果非常准确。

哪里又不对劲了呢？原来，为这次比赛准备数据的人删除了顾客是否购买珠宝的信息，但数据集中只包含了那些买过东西的用户。因此，那些sum(revenue) ＝ 0的顾客，一定是只买了珠宝的顾客。数据集里只包含有过购买行为的顾客，这本身就是一件很奇怪的事，特别是在顾客未完成购买行为前，你无法使用该数据。这个模型不是在正确的数据上训练出来的，没有任何用处。这是一个抽样问题，也是一个经常碰到的问题。

> 关于如何对用户进行抽样的警告
>
> 上面提到的这个例子，仅对有过购买行为的用户进行分析有点奇怪。你是想要将分析仅局限在这批用户，

还是访问网站的所有用户？更一般地，如果你不认真对待手头的用户信息，不把思路理清楚，就有可能犯非常简单但又非常严重的抽样错误。比如想要分析网站一天的用户访问记录，就有可能出现对经常访问网站的用户过采样的问题。

用一个小例子来想一下这个问题：假设有80个用户，其中10个每天都来访问你的网站，剩下的一周只访问一次，假设他们的访问时间均匀地分布在每周的7天里，那么任意一天都会有20个用户访问你的网站。这其中的10人是每天都来的用户，另外10人是每周来一次的用户。这里就会每天对访问网站的用户进行了过采样。他们访问网站的行为也许和其他用户有着根本不同，虽然他们是整个用户数量的12.5%，但却占据了采样数据的50%。

13.4.4　IBM客户锁定

在IBM，需要预测哪些公司有意向购买websphere解决方案，数据是一些交易记录，通过网络爬虫抓取潜在公司的主页得来。一个得奖的模型指出，如果websphere出现在公司的主页上，那么这家公司购买websphere解决方案的可能性就非常大。哪里有错了呢？要记住，潜在客户的定义是那些还没有购买产品的公司（否则，IBM就不会试图将产品卖给它），因此，没有潜在客户会在它的主页上显示websphere相关信息，这根本就不是一个预测变量。如果IBM回到websphere解决方案还未诞生的年代，能看到网站当时的快照，以此作为数据源，那么这个预测变量是有意义的。遗憾的是，现在的数据已经包含了泄漏的信息，它们已经购买过websphere了。再说，你也无法抓取过去的网页，只能抓取现在的网页。

这看起来是一个愚蠢并且显而易见的错误，谁也不会犯这种错误。也许如此，但是这种事时有发生，而且在深入理解数据、弄明白特征和预测变量的含义前，你无法预料到这种事情会发生。想一下，如果这种“显而易见”的错误都会犯，那么对于那些不那么明显的情况更应该加倍小心。同时，这也是在本书中没有给予充分强调的一个例子，与网络爬虫和洋气的机器学习算法相比，通过一些基本的检查，确保一切如你所料，往往会让你走得更远。这可能看起来不够酷，不够吸引人，但它管用，是一个很好的经验。人们不会在聚会时谈起它，也不会将其作为研究成果发布出来，但是它是合理的，而且管用。（不过话又说回来，Claudia因为使用了这个技巧，赢得了多次比赛，而且

经常被邀请参与各种聚会。所以我们收回刚才的话。不，我们不用这样做。关键是把事情做好，其他人自然会跟随你。聚会和名声并不是目标，目标是追求真理。）

13.4.5　乳腺癌检测

假设要研究哪些人患有乳腺癌。看看图13-1，患者的ID看起来无足轻重，却是预测模型中很重要的一个变量。看看发生了什么？

在图13-1中，红色代表患有乳腺癌的患者，绿色代表没有患乳腺癌的人，根据患者的ID做了一个散点图。很容易发现，病人根据患者ID划分成了三四个明显的区块。不同的区块在这里有很强的预测能力。这可能是因为数据来自不同癌症治疗中心的数据库，有些治疗中心专门用于救治那些重病患者——顾名思义，来这个中心就诊的患者得癌症的几率更大。

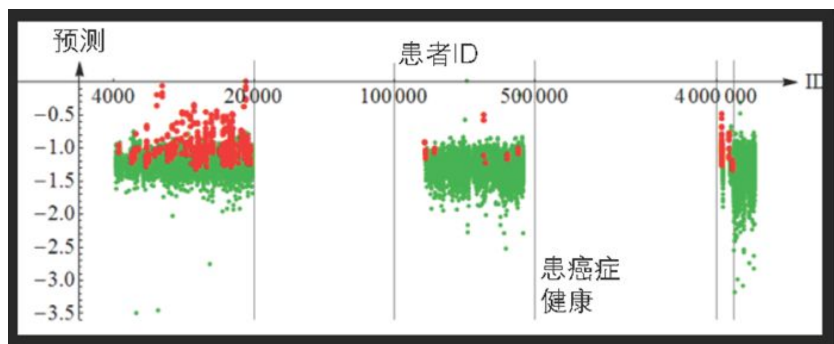


图13-1：依患者的ID排序，红色代表得癌症的患者，绿色代表未得癌症的人

这种情况在课堂上引起了一场有趣的讨论。

学生甲：为了使比赛公平公正，应该给患者重新随机编号。

Claudia：这能解决问题吗？还会有其他类似属性存在。

学生乙：这取决于除了病人的ID，还有哪些属性可以用来判断病人来自哪里。

Claudia：不妨这样想想：我们究竟要使用模型做什么？怎么样才能真正预测出哪些患者患有乳腺癌？

假设现在有一个新来的病人，你将怎么做？如果这个病人的ID落在了

第五个区块里，那这种根据病人ID预测的方式显然不靠谱。但是如果不是这种情况，则使用病人ID做预测就是一种很好的方法。

这个讨论将我们带回到一个基本的问题：我们需要知道建立模型的目的、怎么使用模型和模型是否奏效，以帮助我们决定如何构建模型。

13.4.6 预测肺炎

在一次INFORMS竞赛中，题目是根据病历预测病人是否患有肺炎——将诊断码当作一个数值型变量时，预测准确率不高（曲线下面积为0.80），如果将诊断码作为分类变量，则准确率可以提高至0.90。这是为什么呢？

这和本次比赛的数据准备方式有关，如图13-2所示。

icd1x	icd2x	icd3x	icd4x
786	285	459	-1
401	486	-1	-1
401	486	780	-1
599	-1	-1	-1
V22	650	-1	-1
V56	492	586	-1
786	493	285	459

→

icd1x	icd2x	icd3x	icd4x
786	285	459	-1
401	-1	-1	-1
401	780	-1	-1
599	-1	-1	-1
V22	650	-1	-1
V56	492	586	-1
786	493	285	459

图13-2：INFORMS竞赛中数据是如何准备的

肺炎的诊断码为486。所以如果该数据出现了，准备数据时就删掉它（用-1取而代之）。数据中的行代表不同患者，列代表各项诊断结果，最多有四项诊断，-1表示未进行该项目的诊断。

为了避免数据泄漏，在必要时，将其他诊断结果向左进行了平移，这样将-1都留在了右边。

这样做有两个问题：

- 如果某一行记录只包含-1，那么该病人肯定患有肺炎；
- 如果某一行记录不包含-1，该病人肯定没有得肺炎（除非有五项诊断，但这并不常见）。

仅仅凭借这一信息就能赢得比赛。



发生泄漏

比赛中，利用数据泄漏比构建一个好的数学模型更容易获胜。即使你没有觉察到数据泄漏，你的模型也会感知到这种情况，从而不自觉地利用它取得比赛的胜利。无论怎么说，数据泄漏都是数据挖掘竞赛中面临的大问题。

13.5　如何避免数据泄漏

这里并不是要告诉你如何利用数据泄漏来赢得各种数据竞赛。作为一个数据科学家，在准备数据、清理数据、补偿缺失值和移除异常值等过程中，总会存在数据泄漏的风险。在准备数据时，你可能会无意中扭曲数据，让模型在这份所谓的“干净”的数据上表现良好，但将该模型应用于真实场景中时，效果却糟透了。Claudia给了我们一些避免数据泄漏的具体建议。首先需要暂时去掉那些事先已经知道的信息，比如在一个患者确诊前的信息。每一条记录都应该有一个时间戳，记录你得到该信息的时间，而不是这条记录发生的时间。去掉那些制造麻烦的行和列，特别是那些很容易发现的不一致的信息。经过深思熟虑，从一开始就使用干净的原始数据，未尝不是一个很好的实践。最后，你需要知道数据到底是如何产生的。

在前面提到的文章中，Claudia和她的合作者提出了避免数据泄漏的一个“两步走”方法：首先在收集数据阶段，为所有记录打上合适的标签，然后执行一个他们称作“学习和预测分离”的过程。

13.6　模型评价

怎么评价一个模型好不好？我们已经在前面一些章节中讨论过这个问题，但是，从专家那里听取一些建议总是一件好事。

在使用强大的算法寻找模型中的模式时，存在过拟合的风险。这个概念有点难理解，但是它的基本含义是“只要你观察得足够仔细，总能发现点什么”，即使这种从训练数据上得到的结论不能推广到一般情

况。

为了避免过拟合，可以采取交叉验证和简化模型的方式。图13-3展示了一种标准的情形（需要注意的是，实际工作中一般面对的是高维度空间，通常没有这样的图形可看）。

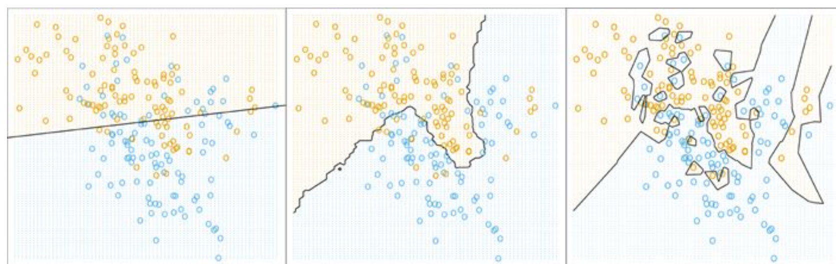


图3-3：这幅经典的图片来摘自Hastie和Tibshirani合著的*Elements of Statistical Learning*（《统计学习基础》），Springer-Verlag，参见http://stanford.io/17szrYz），展示了同一份数据，对二值响应拟合线性回归模型时，采用15个最近邻和1个最近邻得到的不同结果

左图有点欠拟合，中间一幅刚好，右图过拟合。

将过拟合考虑在内，选择哪种模型差别很大，如图13-4所示。

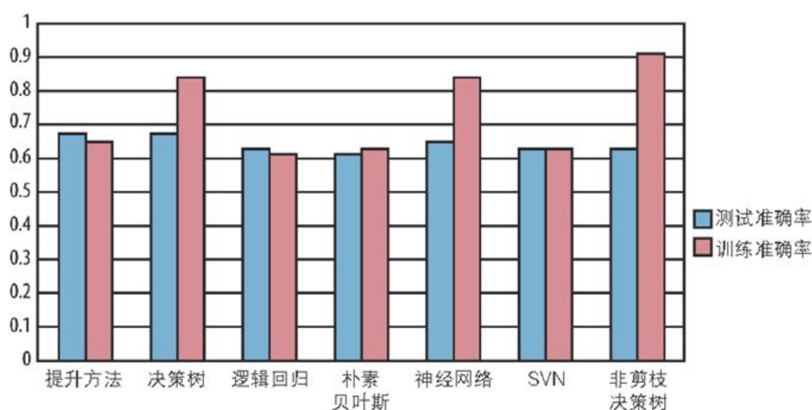


图13-4：模型的差别

仔细看图13-4，非剪枝决策树是“最过度拟合”的（“最过度拟合”是我们现发明的词），这是非剪枝决策树中广为人知的一个问题，因此，人们通常使用剪枝决策树。

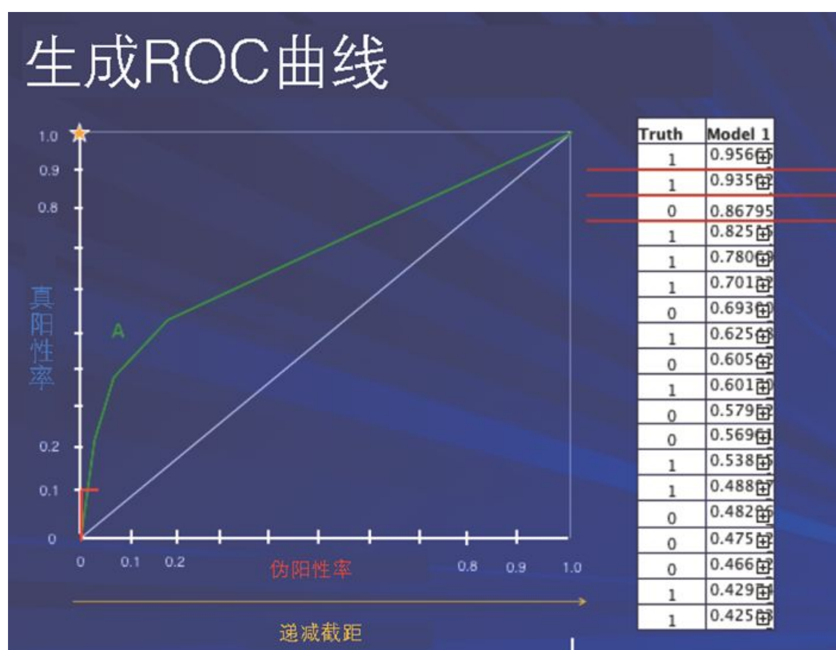
13.6.1 准确度重要吗

在本书的讨论中，评价模型好坏的标准之一是准确度，尤其是对于那些二元分类问题。Claudia认为准确度不是衡量模型的一个好的指标，使用准确度有什么问题？首先，它显然不适合用来评价回归模型；其次，对于那些大部分输出为1的二元分类模型也不适合，一个很傻的模型可能拥有很高的预测准确度，却不是一个好的模型（它预测所有的输出都是1），一个好的模型却可能拥有较低的准确度。

13.6.2 概率的重要性，不是非0即1

没有人会根据二元输出做决策。你想知道的是罹患乳腺癌的概率，而不是一个是或不是的答案。概率包含了更多的信息。人们更看重概率。

那么Claudia认为评价一个模型好坏的标准是什么？她支持将排名和标定分开评价。为了评价排名，可以使用ROC曲线计算曲线下面积，通常是位于0.5和1.0之间的值。这是独立于标定的。图13-5展示了如何绘制ROC曲线。



Truth	Model 1
1	0.956
1	0.935
0	0.86795
1	0.825
1	0.780
1	0.701
0	0.693
1	0.625
0	0.605
1	0.601
0	0.579
0	0.569
1	0.538
1	0.488
0	0.482
0	0.475
0	0.466
1	0.429
1	0.425

图13-5：一个绘制ROC曲线的例子

评价排名时，有时还可以使用升力曲线，如图13-6所示。

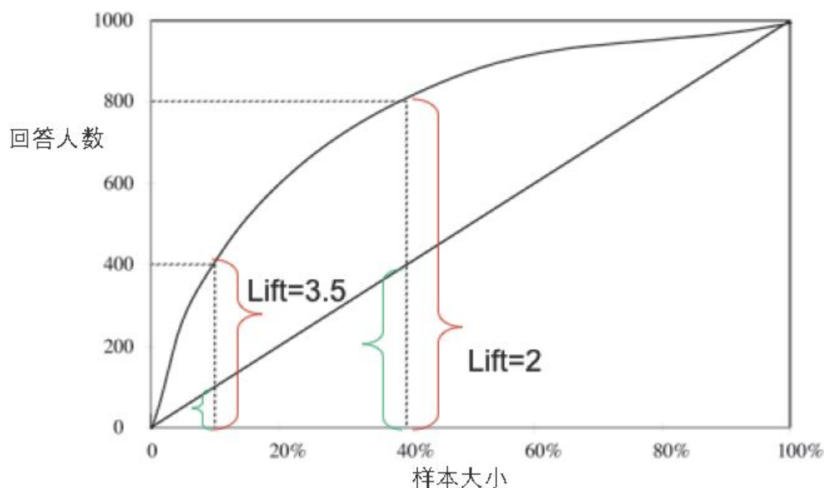


图13-6：升力曲线

升力曲线的关键是升力是基于一个基线计算出来的。对于给定的一个点，比如10%，想象一下，给10%的用户播放广告，对比随机选取10%的用户播放广告，哪种情况用户的点击量更多。升力为3表示点击量多3倍。

如何评价标定？概率准确吗？如果模型预测罹患癌症的概率为0.57，怎么知道概率真是0.57？我们无法直接衡量这个结果，只能将这些概率合并到几个分组中（比如0.50-0.55），然后将它们和实际值进行对比。

图13-7展示了使用非剪枝决策树模型时的情形，图中蓝色的菱形对应相应的分组。

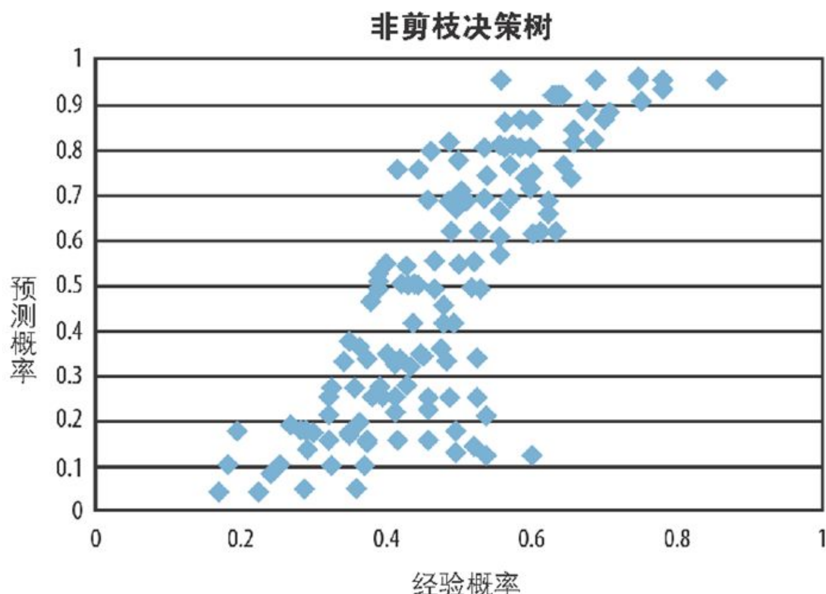


图13-7：评价标定的方法之一是将模型预测得到的概率分组和经验概率分组绘制成散点图进行比较，这里我们使用的是非剪枝决策树模型

蓝色的菱形表示人群；X轴是经验概率，表示经观察确定罹患癌症的人群；Y轴表示利用非剪枝决策树模型预测出来的患病者概率的平均值。该图显示，一般情况下，决策树模型并不是一个评价标定的好标准。

一个好的模型中，分组应该分布在$x = y$曲线附近，但是这张图显示预测概率明显高于实际概率。为什么决策树模型中会有这样的问题？

Claudia解释道，这是由于决策树模型追求优化纯度的特性导致的，决策树里非0即1。因此，相比现实情况，使用决策树模型得到的预测往往显得更极端。这是决策树的普遍问题：它们不适合评价标定。

而Lotistic回归就好很多，结果通常如图13-8所示。

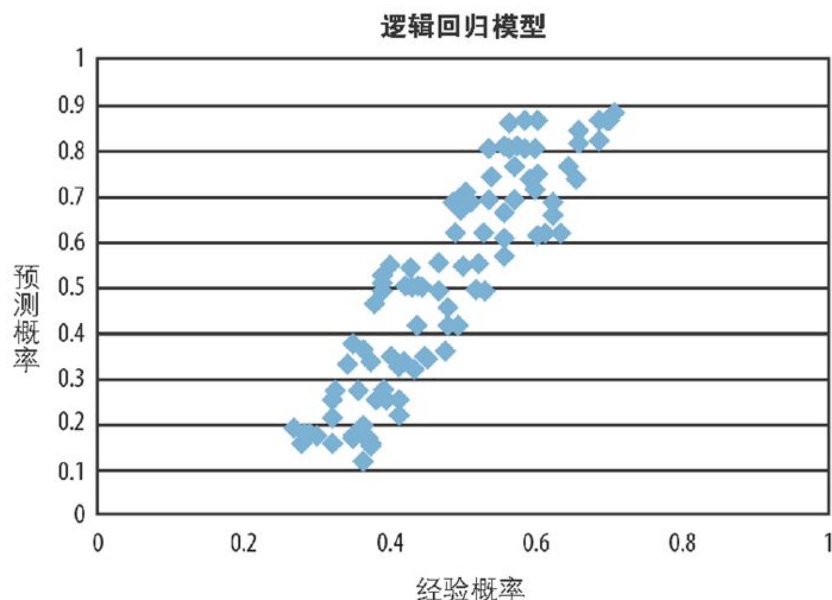


图13-8：测试逻辑回归模型的标定

蓝色的菱形依然表示人群。上图显示了逻辑回归模型在评价模型标定方面做得更好。

13.7　如何选择算法

这并不是一个容易回答的问题，特别是基于小数据集所做的测验往往会让你误入歧途。最佳算法会根据样本容量而发生变化。决策树通常表现不错，但前提是拥有足够多的数据。

一般情况下，需要根据数据集的大小和性质选择算法，选择评价模型的方法则要根据数据和你对模型的期望，你希望它在哪方面表现良好。如果数据服从正态分布，那么误差平方和就是最大似然估计的损失函数；如果要估计中位数，则应该使用绝对误差；如果要估计分位数，则应使加权绝对误差最小化。

我们曾经参加过一个比赛，预测电影在下一年的打分情况，我们假定打分情况服从泊松分布。这时，我们采用的评价方法就不包含让误差平方和最小化，而是包含了泊松分布中特有的属性，该属性依赖唯一的参数λ。因此，有时候需要根据自己的情况，挑选那些适用的评价方式。

13.8 最后一个例子

让我们做个总结。

假设你要为慈善活动募捐。通过向邮件列表里的所有人群发邮件，你希望最后能募得9000美元资金。考虑到一般情况下只有5%的人愿意捐款，为了节省开支，你只给这些愿意捐款的人写信。请问，你如何知道哪些人愿意捐款？

如果选择裁剪过的决策树模型（这是一种标准做法），你的收益将会是0，决策树里找不到一个大多数正向反馈的叶子节点。

如果选择神经网络，即使只给那些预期收益大于开支的人写信，也只能得到7500美元的收益。

让我们试着将这个问题分解如下。收到信的人通常会做两个决定：首先，他们要决定是否捐赠；其次，捐赠多少。我们可以使用如下公式为这两个决定分别进行建模：

$$E(\$ \mid person) = P(response = `yes` \mid person) \cdot E(\$ \mid response = `yes`, person)$$

要注意的是，必须确保第一个模型的准确性，因为你真正关心的是决定捐赠的人数，而不仅仅是捐赠数目。因此，对于第一部分，可以采用逻辑回归，对于第二部分，可以在同意捐赠的人里训练模型。

合起来，这种分解的模型能获得15 000美元的收入。通过分解的方式，模型可以很容易地捕捉到有用的信息。如果数据是无限的，一切都没有问题，你也不需要分解模型。但是现实并不完美，你需要结合实际进行工作。

此外，采用分解的方式也多次引入了误差，如果你有理由相信它们是相关的，那这可能会成为一个问题。

13.9 临别感言

Claudia说，人类不是生来就懂数据的。数据处在我们的感官系统之外，只有极少数人对于数字有种天生的敏感。我们大多数人是命中注

定只能理解语言的。

我们也不是生来就懂不确定性的：由于心存各种偏见，我们难以对不确定性做出正确理解，这些都已经被很好地记录下来。因此，从本质上来说，预测未来要比对已发生的事情进行分类难得多。

即使如此，我们仍竭尽所能，小心翼翼地产生数据，一丝不苟地理解问题，确保使用和真实环境接近的数据进行建模，确保朝我们期望的方向进行优化，不断学习，掌握哪种任务适用哪种算法。

第 14 章　数据工程：MapReduce、Pregel、Hadoop

本章由David Crawshaw和Josh Wills合作完成。Rachel在谷歌工作期间，曾和他们一起共事，服务于Google＋项目的数据科学小组，但是他们俩并未一起工作过，Josh Wills离开谷歌加入Cloudera后，David Crawshaw接替他成为新的技术领头人。我们可以称他们为“数据工程师”，虽然这个称谓像“数据科学家”一样语焉不详（甚至很可能被过度解读），但可以这样说，他们是能够处理海量数据的软件工程师。根据第 2 章讲到的数据科学的工作流程，David和Josh在谷歌的职责是收集数据（前后台日志），搭建海量数据管道以存储和整理数据，搭建用于分析、显示、A/B测试，或者说进行数据科学研究的基础架构。

本章我们将看到来自谷歌工程师的关于MapReduce的一手信息，MapReduce本身就是在谷歌开发的，随后涌现出了各种版本的开源实现。MapReduce是一种算法和框架，用来处理渐在工业界流行起来的海量数据。本章的目的是揭开蒙在MapReduce上的神秘面纱。MapReduce现在都成了时髦术语，很多招聘数据科学家的职位描述上都要求应聘者“必须掌握Hadoop”（Hadoop是MapReduce的一个开源实现）。我们怀疑这些招聘信息是人力资源部的人写的，他们根本就不知道MapReduce是用来干什么的，他们也不知道不是所有的数据科学问题都要用到MapReduce。但是既然MapReduce已经成为一个数据科学的术语，我们想讲清楚它到底是什么、它从哪儿来的。你应该了解MapReduce，但是未必会用到它，这完全取决于你的工作内容。

> 数据科学家需要知道**MapReduce**吗？
>
> 来玩一个好玩的游戏：去参加一个数据科学的会议，数数“MapReduce”被提及了多少次，然后向他们提问什么是MapReduce，看看有多少人能解释清楚。我们怀疑，即使这些参会的专家在工作中已花费数不清的时间来应用MapReduce，但真正能解释清楚的人也不会太多。在谷歌工作期间，Rachel使用Sawzall语言写了一些代码来处理和整理数据，以便进行分析和搭建原型，Sawzall语言将MapReduce框架内建于它的逻辑结构之中。Cathy也曾在Intent Media担任

> 数据科学家期间使用过Pig——一个Sawzall的开源版本，她在Mortar Data框架中同时使用了Pig和Python。我们确实间接地用过MapReduce，我们也确实理解它，但是肯定没有这些亲自实现MapReduce的家伙理解得透彻。

讨论MapReduce的另一个原因是，它说明了在大数据情景下，解决工程和基础架构问题时所使用的一类算法。这是我们在第3章中提出的第三类算法（其他两类是机器学习算法和优化算法）。为了对比，我们还介绍了另一种数据工程算法和框架——Pregel（Pregel也来自谷歌，而且已经开源），它对工程师的算法知识要求不高，支持大规模图的计算。

14.1　关于David Crawshaw

David Crawshaw是谷歌的一名软件工程师，他曾经使用了一段错误的Shell脚本误删了10PB数据，幸运的是，他有备份。David学的是数学，曾和Rachel一起在加利福尼亚为Google＋项目工作，现在他的工作是为更好地理解搜索搭建基础架构，最近他从旧金山搬到了纽约。

David为我们介绍了MapReduce，以及如何处理海量数据。在深入这些内容之前，让我们先做一个思维实验。

14.2　思维实验

我们应如何看待开放查阅病历记录和保护隐私之间的关系？

一方面，将病历记录开放查阅会带来严重的隐私侵犯问题——我们不想让随便一个人都能得到用户的医疗历史。另一方面，有时候获取这些信息则可以挽救患者生命。

据估计，在一个相当小的镇上，每周都有一到两个患者死于信息匮乏，因为其医疗记录不能及时在医院急诊室和附近的精神卫生诊所流通。换句话说，如果这些记录可以非常方便地匹配起来，那么将会有更多的生命得到挽救。另一方面，如果匹配这些记录很容易，一些保密的信息则有可能泄漏。当然，我们很难知道具体有多少生命因这个原因而危在旦夕，但绝不是个小数。

这就自然引出一系列问题：医疗记录里有多少信息属于隐私，是需要保护的；谁有权力访问你的医疗记录；在什么条件下才能访问。

我们假设尊重隐私一般情况下是件好事。比如，无神论者在某些地方是要被判处死刑的，在这种情况下，最好保护这些隐私。但有时候隐私也会带来死亡，就像我们前面讲到的急诊室里因缺乏这些信息导致患者死亡的案例。

让我们再来看看其他例子，执法机关时常要面对一些违法犯罪行为。比如说，执法人员手上掌握着大量的汽车牌照数据，如果被别有用心的人有权限访问，他们有可能会滥用这些信息。在这种情况下，就不是技术的问题，而是人的问题。

这同时也是一个哲学问题：在何种程度上我们可以代替他人做决定？

这里还有一个主观意愿的问题，拥有更多的医疗数据我们或许可以更快地治愈癌症，但是如果患者不愿公开病史，我们也不能因此拒绝治疗他们。

最后，对个人来说，这也是一个安全问题。通常情况下，人们不在意别人拥有这些数据，人们在意的是这些数据是否会给他们造成伤害，或者将数据和他们个体关联起来。

再回到技术问题，搜集数据要做到完全匿名是非常困难的。Alexandre de Montjoye等人在《自然》杂志上发表了一篇文章“Unique in the Crowd: the privacy bounds of human mobility”（“群体中的独立性：移动化趋势中的人类隐私权界限”），研究发现，对于一份包含150万欧洲用户手机号码的数据，仅仅四位数就可以区分95%的人。

最近我们发现人们在竭力反对NSA收集美国公民数据的行为（更别提搜集非美国公民的数据了）。事实上，在本书就要付梓的时候，Edward Snowden爆出了“棱镜门”。这引发了一场广泛且激烈的大辩论，各方激辩政府权力与公民隐私之间的边界。想想现在有多少个人信息通过信息数据仓库和Acxiom这样的经纪公司在网上售卖，这些信息不光包含市场信息，还有保险、职业、贷款信息。我们或许也应该就这些问题和企业展开类似的对话。

14.3 MapReduce

让我们先来看看David是如何站在工程师角度来看待大数据的。

大数据更多意义上只是一个时髦术语，但是它是有用的。David试图用如下的话定义大数据：

当你处理数据时，如果一个计算单元容纳不下这些数据，则这就是大数据。这是一个随时间而演进的定义，按照这个定义，大数据已经存在很长时间了。早在电脑发明之前，美国国家税务局就在征税，根据刚才的定义，这些税收数据也无法放在一个（不存在的）计算单元里，因此也可以叫作大数据。

现在，大数据指不能使用一台计算机处理的数据，即使如此，大数据的数量也在快速地增长。计算机的处理能力在过去四十年中呈指数级增长，现在至少还可以以这个速度持续增长十年（十年前人们就这样说）。

既然这样，大数据会消失吗？我们可以忽略它吗？

David认为我们不能这样想。因为虽然计算机的处理能力也在保持指数级增长，但是这些计算机同时在以同样的处理能力产生数据。新的数据也在以指数级的方式增长。因此有两条指数级增长的曲线，短期内还看不到相交的可能。

让我们用一个实例来看看情况会变得多么复杂。

14.4　单词频率问题

假设要从下述列表中找出出现频率最高的词：red, green, bird, blue, green, red, red。

最容易的方式当然是直接通过肉眼观察，但是假如这个列表变长了，变成包含10 000个、100 000个、1 000 000 000个单词的列表，这个时候怎么办？

最简单的方式是写程序将所有单词列入其中，计算每一个单词出现的次数。图14-1展示了使用Go语言（http://golang.org/）编写的代码片段，Go是David喜欢的语言，他在谷歌参与了该语言的设计与实现（你实现过哪种语言吗）。

```
func count (words []string) {
        counts:- map[string]int{}
        for _, word:= range words{
                counts[word] +=1
        }
        printSorted(counts)
}
```

图14-1：使用Go语言实现的代码片段

计数和排序是很快的，因此这个算法有能力处理包含1亿数量级单词的列表。问题的瓶颈在内存——试想一下，列表中的单词需要两次被装载进内存中：装载列表时一次，为每个单词计数时一次。

可以对程序做一些改进，不必一次装载整个列表，将列表存放在磁盘里，使用管道代替列表，在需要时以流的方式读入。管道像是一种流：读取排在最前面的100个单元，处理完后再读取后100个。

但是这仍然没能解决潜在的问题，如果列表很长，且列表中的每个单词都不同，内存还是放不下，程序依然会崩溃。另一方面，这段程序可能在绝大多数情况下都可正常工作，因为大多数情况下总会有重复的单词出现。写程序就是这样一种混乱的游戏。

先别急，现在的计算机不都是多核的吗？让我们把它们全用上！这时，带宽又成了问题，让我们再把输入进行压缩。不能说这些方法不管用，但是它们增加了复杂性，还有比这些更好的方法，但是复杂性更高。存储散列值的定长的堆表现就不错。堆是一种数据结构，有点类似半排序的集合，可以扔掉那些极小的单元，以避免将所有东西都放在内存里。这种方法并非每次都行得通，但大多数情况下是奏效的。



还能跟上吗？

你不必清楚这里的细节，我们只是想让你知道为什么需要MapReduce。

现在，使用一台计算机可以处理10兆数量级的单词。假设现在有10台电脑，那么可以处理100兆数量级的单词。每台电脑处理1/10的单词，然后将结果汇总到一台“主控”电脑。主控电脑负责将结果汇总，求出出现频率最高的单词。

如果我们学过网络编程，也可以使用存储散列值的堆来解决这个问题。

假设现在有100台电脑，那么就可以处理**1000**兆数量级的单词。不过问题又来了，由于带宽有限，将每台电脑的处理结果发送到主控电脑

时又行不通了。这时需要一个树形拓扑结构，以10台电脑为一组，将结果发送给一个局部的主控电脑，然后这10台局部的主控电脑再发送到最顶级的主控电脑，问题可能就解决了。

但是，这样的方法扩展到1000台电脑时是否同样有效？答案是否定的，这种方式行不通。这么多电脑，总有一两个会坏掉，假设用X表示一台电脑是否工作正常，$X = 0$表示工作正常，$X = 1$表示不正常，那么所有电脑工作正常的概率为：

$$P(X=0)=1-\epsilon$$

这同时意味着，这1000台电脑中没有任何一台工作不正常的概率为：

$$(1-\epsilon)^{1000}$$

即使ϵ足够小，这个概率也是非常小的。假设电脑出错的概率$\epsilon = 0.001$，那么所有1000台电脑全部工作正常的概率为0.37，还不到一半。这种架构显然不够坚固。

那么我们应该怎么做？

想要弄明白这个问题，先得说说分布式系统的容错功能。这通常需要对输入进行备份（默认将输入复制成三份），将每个备份交给不同的电脑处理，如果一份数据处理时损坏了，另外一个备份依然能保证数据完整。还可以将校验码嵌入数据中，这样数据本身就具有了校验错误的能力，我们就可以用一台（或者多台）电脑来实现自动化管理。

一般来说，需要开发一个可以探测到错误、并能自动恢复的系统。为了提高效率，当某些机器执行完任务后，可以重复执行其他任务，当然，免不了还要检测错误。



问：等等，我记得我们是在说计数。现在，我觉得我们又惹上了另一个麻烦。

答：事情总是这样的，论证容错性的效率并非易事，

所有的事情都很复杂。要注意的是，效率和正确性同等重要，你的薪水可比1000台电脑值钱。

看起来像是这样：

- 对付10台电脑很容易；
- 对付100台电脑有点难；
- 对付1000台电脑简直是不可能完成的任务。

没有一点办法！

起码，在8年前是这样。现在，David在谷歌动辄就使用10 000台电脑组成的集群。

初涉MapReduce

2004年，Jeff和Sanjay联合发表了一篇论文“MapReduce: Simplified Data Processing on Large Clusters”（“MapReduce：大集群上简化的数据处理”），同时还有另外一篇论文“The Google File System”（“谷歌文件系统”，参见http://research.google.com/archive/gfs.html ）描述了MapReduce架构的底层文件系统。

MapReduce是个平台，在这个平台上，程序员不用再考虑处理容错性，平台本身就为我们提供了这种服务。它也是一个函数库，用它可以做很多过去不敢想的事。现在，使用MapReduce，在1000台电脑上编程反而比过去在100台电脑上还简单。

使用MapReduce需要写两个函数：一个mapper函数，一个reducer函数。这两个函数会在分布存储数据的各个电脑上运行，只要你将算法放进这种map/reduce的框架，系统就自动具备了容错的功能。

mapper函数将每个数据点作为输入，按顺序输出形如(键, 值)这样的二元组列表，然后框架会使用定向冒泡排序对输出排序，如果键值相同，会将两个二元组合并，将值堆叠起来。这些堆最终会被送往执行reducer函数的机器。reducer函数使用聚合函数对值进行聚合，得到新的值，输出新的(键, 新值)列表。

来看看如何使用这种方式解决上面提到的单词计数算法。以每个单词为键，值设为1，则有如下的二元组：

```
red ---> ("red", 1)
blue ---> ("blue", 1)
```

```
red ---> ("red", 1)
```

然后对其排序，得到一堆("red", 1)，记为("red", 1, 1)，然后被送往reducer函数，reducer函数将所有的1相加，最终结果为：("red", 2), ("blue", 1)。

这里的关键是：一个reducer处理给定键的所有值。

如果数据变得越来越多怎么办？只要增加mapper和reducer的数量就行了。换句话说，使用更多的电脑。MapReduce抹平了在多台电脑上工作时的复杂性。它是如此优雅，以致人们在不需要的时候也用它（然而在谷歌，假设数据每晚呈100倍地增长，这种假设并不算疯狂）。像所有的工具一样，MapReduce被滥用了。

计数本身是一个简单的函数，现在被拆分成了两个函数。一般情况下，如何将一个算法拆解成一些列的MapReduce步骤并不是那么直观的。

对于上面提到的单词计数问题，单词必须服从统一分布。如果所有的单词都一样，那么在排序阶段都会汇入同一台机器，这是个大问题。谷歌使用另外一种数据结构“散列存储桶堆”解决了这个问题，在每个MapReduce迭代的mapper函数中使用该数据结构。谷歌给它起了个名字，叫“CountSketch”，专门用来处理奇异数据集。

在谷歌，有一个监控器实时监控MapReduce作业的状态，绘制成一个条状图，每一条和机器上的一个任务对应。如果每个mapper函数运行正常，条状图看起来就像一条直线。但是通常情况下，这种现象并不多见，由于数据并不服从统一分布，一个键对应太多的值，在reduce阶段，所有的事看起来都不对了。

后台需要运行数据准备和写文件，这通常要花很长时间，因此最好在一个迭代里将所有事做完。这里假设分布式文件系统已经就位，我们需要使用MapReduce将数据写入分布式文件系统——上了MapReduce的船就下不来了。

当来到优化阶段，为一点细小的性能提升，比如在处理PB级的数据时将性能提升几微妙，可能要绞尽脑汁。提升几微妙，这通常是物理学家才会如此关心的事。这部分优化是用C＋＋实现的，这是一段高度优化的代码，我们竭尽所能将系统的性能发挥到极致。

14.5　其他MapReduce案例

单词计数只是MapReduce最基本的案例之一，为了对MapReduce有一个更全面的认识，让我们看看其他一些例子。能用MapReduce解决的问题有一个重要特征：数据可以被分布到多台计算机上，且算法可以在每台计算机上独立处理数据，即每台计算机之间是相互隔离的，它们不需要知道其他机器在处理什么。

下面是能使用MapReduce的另外一个案例。假设现在有记录用户访问某网站行为的数以万计的数据，对于每个用户，有如下形式的一行记录：`{user_id, IP_address, zip code, ad_they_saw, did_they_click}`。假设你想知道不同行政区域（根据不同的邮政编码）看到广告的用户数量，和看到广告并且至少点击一次的用户数量，而且不能将一个用户重复计算。

怎么使用MapReduce处理上述问题？在执行MapReduce作业时，以邮政编码作为键，假设一个用户所在区域的邮政编码为90210，如果他看见广告并且点击了，则有`(90210,{1,1})`，如果他看见了但是没点击，记为`(90210,{0,1})`。

这能带给你什么呢？在reducer阶段，会以邮政编码为组，计算用户看到广告和看到广告并且点击的次数，产生形如`(90210, {700,15530})`这样的输出。但是，这不是题目要求的，题目要求用户不能重复计算。这需要两个MapReduce。

第一个MapReduce使用`{zip_code,user}`作键，`{clicks, impressions}`作值，比如`({90210,user_5321},{0,1})`或者`{90210, user_5321} <- {1,1}`，reducer会根据每个用户、每个邮政编码生成一个包含点击次数和看见次数的表格，形如`{user, zipcode, number_clicks, number_impressions}`。

然后计算来自每个行政区域的不同用户的数量，他们至少点击过一次广告。这时需要第二个MapReduce，以邮政编码作键，`{1, ifelse(clicks>0)}`作值。

这是使用MapReduce计数的另一个演示，但是MapReduce能做点更复杂的事吗？比如实现一个统计模型，线性回归什么的。这可能吗？

答案是肯定的。2006年发表的一篇论文阐述了如何使用MapReduce实现各种类型的机器学习算法（http://machinelearning.wustl.edu/mlpapers/paper_files/NIPS2006_725.pdf）。有些算法在计算各种统计量和变化率时需要先计算期望值和求和，此时就可以使用该论文描述的方法。因为这些计算可以批量处理，可以表示为不同数据点上的

和。

MapReduce不能做什么

有时候，了解一件事是什么，可以帮助了解它不是什么。那么，什么是MapReduce不能做的呢？不能做的事太多了，比如给我们发个消息。认为所有和数据相关的问题都能使用MapReduce的方式解决，这也是可以谅解的。

但是MapReduce并不适合迭代式的算法——计算时将上一次的输出作为下一次计算的输入，很多机器学习算法都如此，它们在计算最速下降收敛方法是都用这样的方式。如果你非要坚持使用MapReduce，这仍然是可能的，但这需要在引擎里四处修改。有一些新的方法更适合处理这种情况，比如Spark，它执行起来效率更高。

14.6　Pregel

为了和MapReduce对比，我们再介绍另一种处理大数据的算法：Pregel。该算法同样出自谷歌，它是一种基于图的计算方法，这里可以将数据想象成一种图状或网状的数据结构，连通的节点之间可以互相交换信息。同时还有聚合节点，可以获取所有节点上的信息，然后在其上进行求和或计算平均数之类的运算。

该算法的基础是运行于节点之间的一些超级步骤，它将信息从一个节点送往另一个节点，从一般节点送往聚合节点。该算法的论文在网上可以找到（http://dl.acm.org/citation.cfm?id=1807184），同时该算法还有一个叫Giraph的开源实现（http://giraph.apache.org/）。

14.7　关于Josh Wills

Josh Wills是Cloudera的数据科学主管，带领工程师为来自各个行业的客户提供基于Hadoop的解决方案，稍后会介绍更多关于Cloudera和Hadoop的内容。在加入Cloudera之前，他为谷歌工作，在那里开发广告竞拍系统，后来领导开发了Google+项目的数据分析基础架构。他在杜克大学取得了数学学士学位，在奥斯汀的田纳西大学取得了运筹学硕士学位。

Josh Wills因对数据科学发表的一些精辟格言而闻名，比如，他曾说“我让数据发光发热”，本书开始也引用了他对数据科学家的定义“数

据科学家(名词)，是软件工程师里最懂统计的，统计学家里最会编程的”，还有“我是阿甘，我有一柄牙刷，我有很多的数据，我的工作就是对着它没日没夜的刷”。

Josh Wills以一个思维实验引入了他的主题。

14.8　思维实验

如何建造一架人力飞机？你会怎么做？你应该如何组建一个团队？

也许你会搞个有奖竞赛（X prize，参见http://www.xprize.org/）。有人就是这样干的，他们在1950年悬赏5万美元。十年后，才有人拿到了那笔奖金。获胜者的故事值得我们在这里讲述，因为它说明了人们有时候致力于解决的问题本身就是错误的。

最初的几个团队花了几年时间做计划，然后飞机飞上天几秒钟就落下来摔得粉碎。而赢得比赛的那个团队则换了个思路：如何建造一架出事后在4小时内就能恢复的飞机？经过几轮快速的原型迭代，他们成功了，只用了6个月。

14.9　给数据科学家的话

通过对数据科学家日常工作的观察，Josh发现90%的时间都被用来清理和准备数据。在解决问题和获得洞见之间，数据科学家往往选择前者。更确切地说，从一个问题出发，确保它有值得优化的地方，然后并行化处理所有事情。

天资聪颖当然是件好事，但是能够快速学习更好，立即着手实验，立即从中学习到有用的东西。

14.9.1　数据丰富和数据匮乏

大多数人偏向保守，他们选择丢弃那些看似无用的东西，习惯使用有限的数据思考问题。Josh选择保留一切。他是可重复性研究的粉丝，所以他希望能够重现分析过程中的任何阶段。这样做有两个原因。首先，如果犯错，他不用把一切推倒重来。其次，当有新的数据时，很容易集成到当前工作流程中来。

14.9.2　设计模型

模型不免最终沦为疯狂的鲁布·戈德堡机械（Rube Goldberg machine，是一种设计得过度复杂的机械组合，以迂回曲折的方法去完成一些其实是非常简单的工作，例如倒一杯茶，或打一只蛋等），一堆模型的大杂烩。这并不总是件坏事，只要模型能工作，就没有问题。即使以一个简单的模型开始，最终还是会添添补补，变得复杂起来。这种事一再发生，没办法，这就是设计模型的本质。

认清分歧

为模型做优化和为业务做优化是不同的。

比如，Facebook的朋友推荐系统并不用来帮助你甄选那些真正志趣相投的朋友，它是为了让你在**Facebook**网站上花费尽可能多的时间。你想想，推荐给你的朋友是不是都是些迷人的异性？

其他领域又是什么情况呢？在医疗机构，他们研究某种药物的疗效而不是患者的健康，他们通常关心手术是否成功，而不是患者的幸福。

14.10　算算Hadoop的经济账

让我们回过头来再看看MapReduce和Hadoop。2001年，Josh刚从学校毕业，当时存储文件有两种选择：数据库和存储文件管理器，Josh从模式、处理能力、可靠性和花销四个维度对二者进行了对比。

表**14-1**：**2001**年时可选的文件存储类型

	[illegible]	
[illegible]		
[illegible]		
[illegible]		
花储大规模数据时花费高		

现在产生的数据越来越多，其中大部分来自网页。这自然引出了衡量投入产出比的经济学指标：字节价值。从一字节的数据里我们可以获取多少价值？存储它又要花费多少钱？如果我们求两者的比值，则希望这个比值大于1，否则不如丢弃这些数据。

当然，这不是故事的全部。有一个有关大数据的经济学定律：任何单条记录都不是特别有用的，但是拥有所有的记录则价值连城。比如，对于网页索引、推荐系统、传感器数据、在线广告等系统，如果拥有

现存的所有数据是特别有用的，但单独的每个数据点其实是没有什么价值的。

14.10.1 Hadoop简介

在谷歌还没有像今天这样有钱时，它们的硬件简直差劲极了。为了处理数据，他们想出了一个主意，将数据拷贝到多台服务器上。一开始，他们手动拷贝，后来变成自动化的，这个自动化的过程后来发展成全局文件系统GFS。

Hadoop是谷歌的GFS和MapReduce的开源实现（读者可以在其他地方了解到关于Haddop起源的故事，这里我们只给出两点提示：有一个叫Nutch的开源项目，雅虎公司也参与其中），它的核心分成两部分。第一部分是分布式文件系统（HDFS），它是基于谷歌的文件系统。数据存储在大文件里，文件块的大小通常为64 MB~256 MB之间。这些文件块被复制到集群中的多个节点上。如果一个节点宕机，主控节点会得到通知。第二部分是MapReduce，David Crawshaw已经在前面讲过了，这里不再赘述。

Hadoop是用Java实现的，谷歌的那一摊子是用C++实现的。使用Hadoop提供的Java API编写MapReduce程序并不会令人感到愉快，有时候需要写大量的MapReduce。但是，如果使用Hadoop流，则可以使用Python、R或其他高级编程语言。这使得编写并行任务程序变得简单方便。

14.10.2 Cloudera

Cloudera是由Doug Cutting和Jeff Hammerbacher共同创建的，前一位是Hadoop的创始人之一，后一位我们在第1章提起过，在Facebook工作期间，他率先提出了“数据科学家”这个职位，并为Facebook组建了数据科学团队。

Cloudera之于Hadoop就像Red Hat之于Linux，同样都是围绕一个开源项目创立了一家公司。Hadoop是由Apache软件基金会赞助的，代码是免费的，但是Cloudera将所有东西打包，并且免费提供各个版本，它靠为客户提供技术支持和服务赚钱，赚回来的钱反过来又可以反哺项目，使Hadoop得到更好的发展。

Apache Hive（http://hive.apache.org/）是建造于Hadoop之上的一个数据仓库系统，它使用类似SQL的查询语言（包括某些特定的MapReduce扩展），实现了常见的合并和聚合操作。对于精通数据库

和熟悉这些操作的人来说，这是再好不过了。

14.11　Josh的工作流程

来看看Josh是如何使用MapReduce打造数据管道的，他将一条记录视为数据分析的基本单元。我们不止一次的提及“打上时间戳的活动数据”，你可以将每一条视作一项记录，或者像前面讨论过的交易记录，比如欺诈检测、信用卡交易。一个典型的工作流程如下：

1. 使用Hive（一种运行在Hadoop上的类SQL语言）将你对某实体的了解情况创建为记录（比如一个人）——这里会使用到大量的MapReduce）；

2. 编写Python脚本反复处理这些记录（速度要快，迭代要快，同时还会用到MapReduce）；

3. 当有新数据时，及时更新记录。

要注意第2步的脚本通常只是map作业，它让并行化变得简单。

Josh更喜欢标准的数据格式：巨大的文本占据了空间，而Thrift（http://en.wikipedia.org/wiki/Apache_Thrift）、Avro（http://en.wikipedia.org/wiki/Apache_Avro）和协议缓存（http://en.wikipedia.org/wiki/Protobuf）是一种更紧凑的二进制数据结构。Josh还鼓励大家使用用来存储代码和元数据的Github（https://github.com/），他不使用Git存储大文件。

14.12　如何开始使用Hadoop

如果你所在公司拥有Hadoop集群，很有可能你和Hadoop的第一次亲密接触是通过Apache Hive，它在HDFS和MapReduce之上，提供了一种类似SQL风格的抽象层。你的第一个MapReduce作业很可能是通过对记录用户行为的日志进行分析，来更好地了解客户如何使用公司的产品。

如果你要使用Hadoop和MapReduce编写自己的分析应用，有很多途径可以上手。使用Apache Mahout搭建一个推荐引擎就是一个很好的选择。Apache Mahout包含了一些机器学习的程序库和命令行工具，可以和Hadoop一起工作。它有一个叫作Taste的协同过滤引擎，使用它，给定一个CSV文件，包含用户ID、物品ID和一个可选的，表征用

户和物品联系紧密程度的权重信息，就可以创建一个推荐引擎。Taste使用的推荐算法和Netflix、亚马逊使用的推荐算法都是一样的。

第 15 章　听听学生们怎么说

“每一个算法都是一篇文章。”

— Emily Bell ，哥伦比亚大学新闻学院数字新闻学塔尔中心主任

我们邀请了最早一批选修了数据导论课程的学生来写作本章，他们分享了对本课的想法和自己的学习经历。参与编写本章内容的同学有：Alexandra Boghosian、Jed Dougherty、Eurry Kim、Albert Lee、Adam Obeng和Kaz Sakamoto。

15.1　重在过程

开始学习数据科学时，你别无选择，只能从最前沿的地方开始，因为数据科学本身就处在科技发展的最前沿。

物理学的入门课程通常从易到难，先介绍经典力学、电磁学，然后过渡到近代物理，比如狭义相对论之类。但是这种循序渐进的教学方式并未揭示物理学原理的发现过程，比如，牛顿究竟是如何发明了微积分？没人教给我们这个发现的过程。牛顿是如何做到的？我们不知道他使用了什么工具，他读了什么书。他记笔记吗？他有没有尝试复现别人的证明？他集中精力解决的问题是否来源于他上次写的某篇文章？到底是什么促使他去想“我必须有极限才能解决这个问题”？牛顿演算时需要打草稿吗？还是说很多想法在他心中早已成型，看见苹果落地时碰巧都涌现出来了？这些东西是教不来的，但又是我们必须学会的，那些初出茅庐的科学家必须学会这个过程。

Rachel在数据科学导论的第一堂课上就严肃告诫我们：无论在学术界还是工业界，数据科学尚在定义之中。在接下来的课程里，我们遇见了实实在在的问题，并且看到老师们是如何在这些问题中选择他们想要研究的课题。实际上，每周的课程都覆盖了数据科学家需要用到的工具和技术，只是每节课都有它自己的风格和背景，解决问题的方式也不一样。几乎在每一节课上，老师们都会说：“我不知道你们之前都学了些什么，但是……”课程是离散的，我们要负责将这些残章断片拼成一个关于数据科学的完整介绍。我们要从课程里发掘出对自己有用的部分，就像数据科学家们在持续不断地建造他们自己的领域。

这并不是说Rachel有意要将我们蒙在鼓里。在课程的第一天，她就提出了一个数据科学家的定义，她说，数据科学家就是兼具如下能力的人：数学、统计学、计算机科学、机器学习、可视化、沟通的技巧和行业知识。随后我们马上发现，就像我们理解的贝叶斯定理一样，这只是些先验知识。同学们和讲师们都根据这个定义，对自己做了评估，为数据科学界提供了多样性的写照，并且这份评估也作为参考，贯穿课程始终。担任该课的讲师们来自学术界、金融业、科技公司和初创公司，他们有的在研究生期间就中途退学，有的则是Kaggle竞赛的获胜者，还有数字艺术家。每个人都提供了更进一步的似然比。课程本身就在对数据科学进行迭代式的定义。

但是我们并不是每周都在课堂上听他们谈论自己的工作。通过完成大量很难的作业，我们学会了数据科学研究所使用的工具。有时候，作业要求实现课堂上讨论的技术和概念，有时候我们则需要自己去发现和探索还不知道的技术。

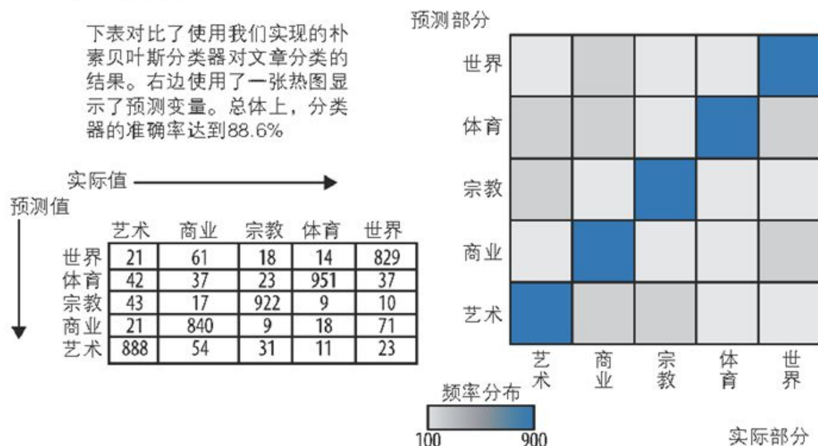
另外，我们面对的是混乱的真实数据。我们经常要去解决工业界内的真实问题，并且需要最终形成一份清晰和深刻的报告。这份报告，即使拿给业界的专家看，我们也会感到自豪。最重要的，为了完成作业，我们经常要跨出自己的舒适区。在这里，强调了数据科学的社会性，作业和项目都需要分成小组，大家合作完成。Rachel还常常带领我们和当日的讲师，穿过街道去酒吧坐坐[1]。就这样，在整个学期里，我们大家一起工作，一起喝酒，互相学习，共同学习从事数据科学工作需要的技能。

> 1 Rachle注释说“这是一门研究生课程，我确保学生们都到了可以喝酒的年龄，而且为不喝酒的同学我们也提供了非酒精类饮品。”

15.2　不再简单

真正的挑战来了，第二个作业（4.7节）的第三小节，要求从《纽约时报》的官方网站上下载2000篇文章，并且通过这些文章训练出一个朴素贝叶斯分类器，按文章出现的不同版面对其进行排序。但是那个网站一次只允许下载20篇文章！抓取这些文章只是万里长征的第一步，我们还要亲自编写代码实现贝叶斯公式，唯一能帮我们的也就是老师给出的几个公式。很多编程语言都有现成的实现，倒不是说这些实现对我们没有帮助，我们的实现有一些特别的需求，我们要求正则化的超参数是可调的，作业还要求将文章分为5类，而不是两类。我们听说朴素贝叶斯并不是那么“朴素”，也不是很“贝叶斯”，它的实现并没有用到贝叶斯模型。结果正是如此，朴素贝叶斯不简单。然而，

我们中的有些人还是坚持下来了，他们在教室里熬了40个小时去打磨一段大概300多行、令人反胃的R代码，最终使模型的预测准确度达到了90%，别提有多高兴了！我们立刻对这件事上瘾了，当然，也可能是我们不忍心丢掉沉没成本——那些在这个作业上花的精力和时间。总之，我们是上瘾了。图15-1是一位同学的答案。



	艺术	商业	宗教	体育	世界
世界	21	61	18	14	829
体育	42	37	23	951	37
宗教	43	17	922	9	10
商业	21	840	9	18	71
艺术	888	54	31	11	23

图15-1：一位同学的部分答案

参加Kaggle竞赛是最后的大作业的一部分，它给了我们另辟蹊径的机会。作业是在学生们中间进行一场竞赛，设计一个论文评分的算法。一般的作业大多模拟工业界的数据科学家们的工作内容，而Kaggle竞赛是数据科学界的一个一决雌雄的比赛，它鼓励我们将课堂上学到的所有关于数据科学的最佳方法都应用上，同时又使得我们可以亲历经典的数据科学活动。本书的作者之一的方案引入了如下特征变量：拼写错误的个数、Dale-Chall单词列表（这是一个四年级学生应该掌握的词汇表）、论文中最常出现的前50个单词的TF-IDF向量、论文单词数的四次方根。别问我为什么，方案使用了随机森林模型和Gradient boosting算法，并且实现了随机超参数优化，在亚马逊提供的EC2平台上运行了几千个小时，训练了5万个模型，它很管用。

15.3　援助之手

在课程的前半程，我们遇见了客座讲师Jake Hofman。你还记得第一次看见魔术时的情景吗？没错，Jake的课就像在你眼皮底下表演魔

术。仅仅用了一些简单的Unix系统命令和数据，他就在我们面前活生生写出了一个朴素贝叶斯垃圾邮件分类器。在黑板上写了一串数学公式后，他现场下载了安然公司的email数据，然后通过高超的命令行技巧分析了这些数据。

在每周一次的课堂上，都有超级棒的讲师们来给我们做讲座。其中，Jared Lander和Ben Reddy主要负责辅导我们的课后实验，在他们的帮助下，我们才得以在这门快节奏的课上不掉队。他们给我们展示了数据科学的结构。我们的课程覆盖广泛，从线性回归到随机森林算法。我们还认识了很多新的工具：正则表达式、LaTex、SQL、R、Shell脚本、git、Python。我们掌握了通过API和网页抓取获取新数据源的关键技术。

来上这个课的人五花八门，每个人都有需要补充的知识。计算机科学家需要快速学会基本的特征选择理论和使用R语言；社会学家需要了解数据库的工作原理，需要知道全局变量和局部变量的区别；金融专业的需要学点道德伦理。大家的学业负担都各不相同。通过慢条斯理地用R写循环（虽然最终发现还写错了），或是思考程序如此低效的原因，我们的知识在一点一点增加。图15-2即是我们从众多教训中学到的一例。

```
#WARNING THIS TAKES A LONG TIME AND MAKES YOUR COMPUTER GET REALLY HOT
detailed_results <- predict(model,as.matrix(testmatrix),type="raw");
```

图15-2：从教训中学到的

随着技能的增加，我们能更多的将精力放在如何分析数据上。最终，我们做到了眼中无代码、心中有想法的境界。

这些是仅凭个人能力就能做到的吗？有人能拿下所有的东西吗？

这需要花费数小时不停地和错误做斗争，才能越过学习曲线，这时我们才领略到了数据科学之美。为了按时完成作业，我们需要互相学习，参加本课的同学大都拥有不同的背景知识。

事实上，为了完成作业，找到知识结构上和自己互补的同学甚至是必须的。Rachel虽然没有要求我们团队协作，但她布置了大量作业，其中不少作业所需要的知识点都比较分散。我们不得不自发组成团队。原来Rachel是要让我们知道，数据科学本质上是一门需要合作的学科。开课之初，Rachel向我们展示了一种类似汽车轮毂状的网络，她让我们绕她围成一圈，辐条将每个人和她连在一起。她希望在课堂上，我们可以建立起新的友谊、提出新的想法、完成新的项目、形成

新的联系。

对于这种刚刚萌芽的学科，参与到社区中去变得异常重要。以数据科学为例，社区不仅利于职业生涯发展，而且对于从事日常数据科学相关活动也很重要。如果你不读相关的博客、关注推特上的人，或者不参加行业内的聚会，你如何能知道最新的分布式软件，或者对于某篇著名文章中用到的统计学方法，业内出现了驳斥声音？社区与我们息息相关，在四月份的一次聚会上，Cathy谈起了MapReduce，她马上可以就一个问题咨询在场的观众Nick Avteniev——他是这方面的专家。在社区里，这样的事经常发生。数据科学的知识体系是不断在变化的，而且分散在不同地方。要知道哪些知识是你应该掌握的，唯一方法是看看别人都知道些什么。邀请不同领域的讲师来讲课为我们开启了这一旅程。所有的讲师都乐于回答我们的问题，他们给我们留了他们的电子邮件地址，有些甚至为我们提供了工作机会。

在听过这些专家的课，并且和他们聊过之后，我们有了更多的问题。怎么样在R里创建一个时间序列对象？为什么为那个该死的矩阵绘制散点图时不断的出错？随机森林到底是什么鬼东西？我们不仅求助于周围的同学，还同时求助于网上社区，比如Stack Overflow、谷歌新闻组和有关R的博客。我们惊喜的发现，有那么多社区帮助像我们这样的菜鸟数据科学家成长，帮助我们让自己的代码跑起来。我们不仅仅从以前遇到类似问题的人那里得到答案，这些答案早就被发明这些方法的先驱们解答过了。比如Hadley Wickham、Wes McKinney、Mike Bostock，都为他们写的程序包提供支持。酷毙了！

15.4　殊途同归

世上并不存在柏拉图式的数据科学知识仓库，让你在里面浸淫一下就能掌握相关知识技能。数据科学中所涉猎的各学科都各有优势，其专业词汇各不相同，有时对同样的方法也有不同的解释（正则化参数是先验概率还是仅仅为了平滑曲线？我们该通过原则性理由还是为了要最大化拟合模型选择参数？）没有什么现成的规矩可循，因为规矩还没定出来，这就是为什么交互作用的结构如此重要的原因：你可以制定自己的规矩。你可以自行选择接受谁的影响，像Gabriel Tarde说的那样（Bruno Latour、Mark Hansen都曾援引他的话）：

> 当一个年轻的农民看到日落时，不知道他是选择相信老师说的日心说——这是地球运动的结果，还是选择接受自己的感觉，认为这是太阳运动的结果。这时，这个年轻人心中仿佛有一束射线，经过他的老师，将

他和伽利略联系在一起。

— Gabriel Tarde

选择站在巨人的肩上没有错，但是在爬上巨人的背上之前，或许应该确认一下巨人是否能承受这一重量。在商业上，使用数据卖广告是个热门话题。你或许有权对世界上最好的数据集进行分析，但是如果有人雇用你仅仅是为了找到多卖几双鞋的方法，这样做真的值得吗？

在我们做作业、对答案时，明显发现不同的决定能导致分析结果千差万别。从做出假设到得到最终结果，即使你掌握了其中所有的分析步骤，但由于每一个步骤中仍然存在多种选择，不同选择的组合将是一个庞大的数字。即使如此，那也不是简单地将一个命令的输出传入下一个命令。算法是可裁剪的，选择哪种算法，使用哪些变量更是如此。

来自Media 6 Degrees（M6D）的Claudia Perlich连续在2003年、2007年、2008年、2009年赢得KDD杯数据挖掘竞赛，现在她是这项赛事的组委会成员。她慷慨地和我们分享了关于数据科学的得失利弊，以及做决定时采取的不同方式。有次竞赛是预测医院的患者治愈率，她发现患者是按顺序编号的，因此，来自同一个诊所的患者都是连号的。不同医院的条件和患者疾病的严重程度都是不同的，因此，患者的编号成了预测治愈率的一个重要指标。显然，将这种数据漏洞包含在内并不是有意为之，它让整个比赛变成了小菜一碟儿。但是在现实中，它应该用在预测模型中，毕竟，医生和病人选择的诊所是应该被用于预测患者治愈率的。

David Madigan强调了在数据科学领域做决策时来自道德方面的挑战，通过对制药业的观察研究发现，预测结果也常常差别很大（他向我们展示了阿司匹林的散点图）。他强调除真实数据之外，不能忽略研究者自身的重要性。仅仅调整模型和方法，并且将它们应用到数据上是不够的。

学术界也面临和工业界类似的问题，尽管出于不同原因。在各学科之间，数据科学的差别是如此之大，以致于不能通过单独对某门学科的研究得出数据科学总体的样貌，这些四分五裂的东西究竟是怎么攒在一起的？它们能攒在一起吗？下面这个例子展示了从纯学术的观点是如何量化问题的，这个例子是一道作业题，来自*The Elements of Statistical Learning*（《统计学习基础》）的“Linear Methods for Regression”（线性回归方法）一章：

练习3.2 已知变量X和Y的数据，试拟合一个三次多项式回归模型$f(X)=\Sigma_{j=0}^{3}\beta_j X^j$，拟合曲线需要达到95%的置信带。试考虑如下两种拟合方式：

1. 对于给定某点$x0$，使线性函数$a^T\beta=\Sigma_{j=0}^{3}\beta_j x0^j$在该点的置信区间为95%；

2. 对参数β求95%的置信集合，反过来可计算出$f(x0)$的置信区间。

这两种方式有什么不同？哪一个置信带的范围更宽？进行一个小型的模拟实验来比较两种方法的优劣。

这是在机器学习或数据挖掘课上一般会布置的作业。作为初出茅庐的数据科学家，我们现在的第一反应应该是怀疑。在数据分析的过程中，什么时候这样的问题才会出现？在问题出现之前我们都做了些什么？为什么我们只考虑这两个变量，而不是其他变量？我们是如何拿到数据的？谁给的？谁为此买单？为什么要计算95%的置信区间？使用其他度量指标效果会不会更好？说真的，谁在乎我们的模型在训练数据上表现得多好？

这对Hastie和他的合作者而言并不公平，他们会说如果学生想学如何抓取和组织数据，他们应该找相应的书来看，这是这门课和其他一般入门级课程的鲜明区别。这门课一直在提醒我们，抛开问题的上下文和决策流程，单纯学习数据科学需要的统计学工具是没有意义的。而且，真实数据经常是一团乱麻，处理起来令人头疼；现实中也没人告诉你到底该选择哪种回归模型。但是，仅仅知道这些是不够的，必须通过亲身实践才能理解，纸上得来终觉浅，绝知此事要躬行。

15.5　逢山开路，遇水架桥

用Michael Driscoll的话说，我们这群初出茅庐的数据科学家不像那些土木工程师一样，对于正在做的事，我们没有一个全局的视野，蓝图并不是永远存在。数据科学家像冒险家，他们知道自己要寻找什么，他们口袋里有相应的工具，可能是一张地图，或者几个朋友。当他们跋山涉水终于到达城堡时，可能公主并不在那儿，但这都不重要，重要的是他们沿途走来，踩过乌龟，吃过蘑菇，而且依然可以吐火。如果说科学是一排排管道，我们不是管道工，我们是倔强的超级马里奥兄弟！

15.6 作品展示

图15-3改进了我们在第1章绘制的数据科学家履历表的图形，图15-4展示了数据科学在各大学间的流行程度，这些图形都基于2012年年底的调查数据。

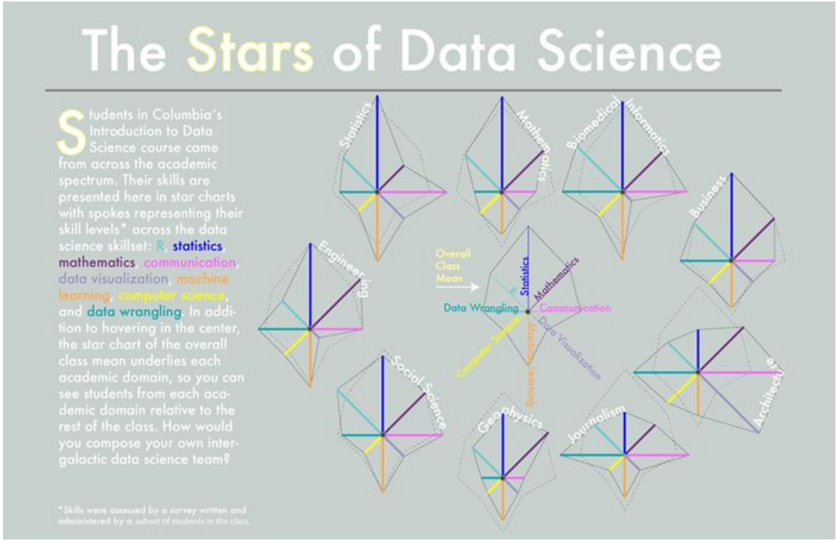


图15-3：数据科学技能星型图（由Adam Obeng、Eurry Kim、Christina Gutierrez、Kaz Sakamoto、Vaibhav Bhandari合作完成）

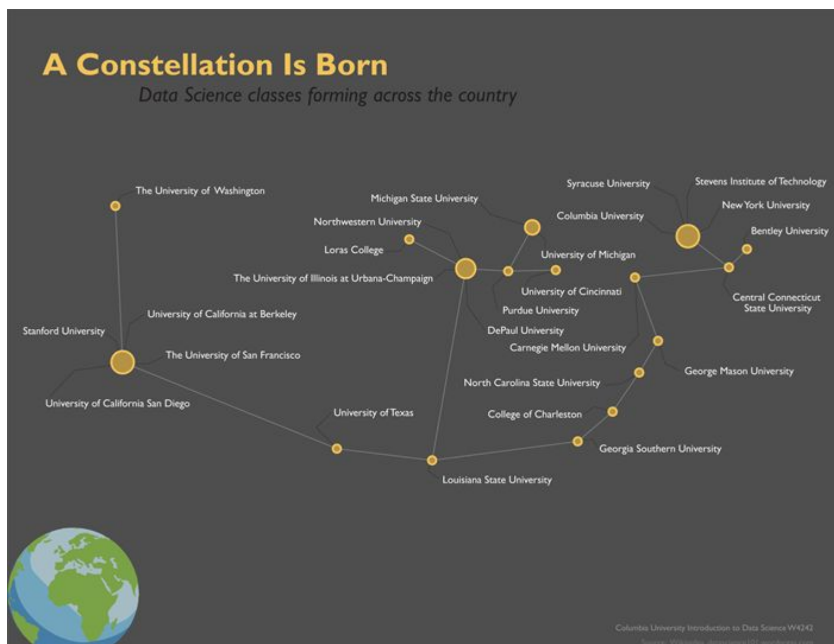


图15-4：数据科学在各大学间流行程度的星座图（由Kaz Sakamoto、Eurry Kim、Vaibhav Bhandari合作完成）

第 16 章　下一代数据科学家、自大狂和职业道德

本章将回顾过去，展望未来，以此对全书做以总结。

16.1　前面都讲了些什么

本书有两个主要目标，一是告诉读者数据科学家是干什么的，二是教会读者一些数据科学家的技能。

我们希望本书已经完成了这两个目标。

针对第一个目标，本书若干章的贡献者为我们带来了大量的关于数据科学家日常工作的一手资料。在本书中，即使没能如我们希望那样全面覆盖数据科学的方方面面，但本书的广度仍然值得我们自豪，这帮助我们实现了第二个目标。

有人可能会写出一本更好的书，下栏里的内容是我们关于这个问题的一点想法。

> 思维实验：如何教授数据科学
>
> 换作是你，你会怎么样去写一本关于数据科学的教材？
>
> 首先，数据科学并没有一个确切的定义，也没有标准可循，它更多地流行于报刊和媒体，但是没有哪家“权威机构”去纠正那些漏洞百出和以讹传讹的错误。其次，数据科学和很多其他学科都有交叉，比如，一本数据科学的教材很容易就会变成另一本机器学习的教材。
>
> 如何衡量一本数据科学教材是否成功？如何衡量它的影响力？反之，如何鉴定一本教材是失败的？
>
> 我们能把这当成一个数据科学问题吗？如果能使用一个因果模型去回答这个问题，那再好不过了。这需要

找到那些和本书读者类似的那些人，但是他们暂未购买本书，然后使用倾向分数配对法进行分析。或者我们可以做个实验，随机挑选一些人，让他们接触不到本书（这有点难，因为人人都可以访问亚马逊网站），而且他们还可以通过其他方法得到。但这都无关紧要，重要的是这比蒙着眼睛猜要强多了。

在业界，一直流传着这样一种说法，数据科学是不可能在大学里或者书本里学会的，必须在工作中边干边学。但这种说法可能不对，这本书可能就是一个证明。学没学会，亲爱的读者，你们说了算。

16.2　什么是数据科学（再问一次）

在书中，我们一次又一次地讨论这个问题，它是本书的主题，是本书的中心问题，都快成口头禅了。

可以简单地通过数据科学家的工作来定义数据科学，在前面讨论数据科学家的知识结构时，我们就这样干过。事实上，在Rachel来到哥伦比亚大学开设这门课程之前，她就列了一个数据科学家的工作项目清单，只是她不愿意拿出来给人看，因为这个清单有点乱，而且条目多得有点吓人。这份清单就是后来提出的数据科学家的知识结构的原始材料。在课程结束后，通过与一些人的交谈，她发现他们希望看到这份清单，于是我们就把它列在下面：

- 探索性数据分析；
- 可视化（用在探索性数据分析和汇报中）；
- 数字面板和矩阵；
- 对业务的洞察力；
- 数据驱动决策；
- 数据工程和处理大数据的能力（Mapreduce、Hadoop、Hive、Pig）；
- 收集数据；
- 构建数据管道（日志→mapreduce→数据文件→同其他数据合并→mapreduce→清除一些噪声→合并）；
- 开发新产品，而不仅仅停留在对现有产品的使用进行描述上；
- Hack；
- 写专利；
- 数据侦探；
- 预测未来的行为或性能；

- 将发现写成报告、做讲演或发表在学术期刊上；
- 编程（要熟练使用R、Python、C和Java等语言）；
- 条件概率；
- 优化；
- 算法、统计模型和机器学习；
- 讲故事的能力；
- 会提问题；
- 做调查；
- 搞研究；
- 从数据中做出推测；
- 开发数据产品；
- 找到处理数据的方法，会根据数据规模改变分析策略；
- 一致性检查；
- 对数据的直觉；
- 和领域专家打交道的能力（或者让自己成为一个领域专家）；
- 设计和分析实验；
- 发现数据间的相关性，并且尝试建立潜在的因果关系。

但是现在，我们想再往前走一步，去追求一些更深刻的东西。

在本书的开头，我们将数据科学定义为在科技企业的一组最佳实践。经过了这么长时间的探讨，现在不妨把眼界放宽，不要把数据科学局限在科技企业里，将其他领域也包括进来：神经学、健康状况分析、电子搜索、计算社会学、数字人文研究、基因学、政治等，将所有可用数据解决的问题包括进来。这些问题都可以利用本书讨论的一些最佳实践来解决，只不过有些实践最早在科技企业建立起来而已。数据科学得以同时在工业界和学术界发展，因此，数据科学应用在哪里或哪个领域并不重要，它的关键是通过算法和代码定义了一套从“题域”到“解域”的映射，而数据是关键中的关键。

因此，我们可以这样定义数据科学：数据科学是一些科技公司的最佳实践，但可推广到所有可以用数据解决的问题，有时叫它科学也无妨。即使这样，有时数据科学也只是沦为一场炒作，这是我们要极力避免的，我们更不应该推波助澜地帮助他们炒作。

16.3　谁是下一代的数据科学家

> “在我的时代，那些最聪明的人都在思考怎么让用户点击广告……这简直弱爆了。”

— Jeff Hammerbacher

理想情况下，这一代正在接受训练的数据科学家不应满足于成为技术高手，在一个舒适的城市找一份薪水不菲的工作，当然这些都不错，但我们应该有更高的追求。我们鼓励下一代数据科学家成为提出问题并解决问题的人，深入思考合适的设计和流程，负责任地使用数据，让这个世界变得更好，而不是更坏。让我们在下面几节详细展开这些内容。

16.3.1　成为解决问题的人

让我们先来讨论数据科学家需要具备的技术。下一代数据科学家应努力具备以下技能：编写程序、统计学、机器学习、可视化、沟通的技巧和数学。同时，有一个坚实的编程基础，良好的编程实践，诸如结对编程、代码审查、调试和版本管理，无疑是很有价值的。

什么时候强调探索性数据分析都为时不晚，这点我们在第2章讲过，同样的，还有Will Cukierski提到的对特征进行选择。Brian Dalessandro强调了数据科学家如何在无数的模型中做出选择—该选用哪种分类器、特征、损失函数、优选法和评价模型的标准。Huffaker讨论了特征或矩阵的构成，从日志中变换变量，构建0-1变量（比如，重复5次同样活动的用户），聚合和计算。虽然这些是数据科学中的关键部分，但经常被认为是微不足道的，因此常常在工作中被忽略了。Dalessandro将这称为“数据科学的艺术”。

另一个警告：很多人拿到数据就直接使用一个花哨的算法。但是在数据和算法之间，还有很长的一段距离。跑一段代码去预测或分类，当算法收敛时就可以宣告取得了成功，这些都很简单。难的是正确地分析和预测，确保结果是正确和说得通的。



下一代数据科学家会怎么做

下一代数据科学家不会试图用复杂但并不奏效的模型来让自己看起来高深莫测。他们花大量的时间整理数据，大概90%的时间，尽管没人愿意承认这一点。最后，他们不会陷入对某种工具、方法或某个学术部门的宗教式崇拜中，他们多才多艺，在各学科间切换自如。

16.3.2 培养软技能

很多人都能实现*k*最小近邻算法，但很多人都没做好。事实上，大多数人开始做得都不是很好，从哪儿开始不重要，重要的是最终能走多远。养成良好的习惯，以开放的心态持续学习是非常重要的。

我们认为下述一些思想品质有助于解决问题1：持之以恒，思考你的大脑是如何思考的，不钻牛角尖，能灵活地思考，永远追求准确度，带有同理心地去思考问题。

> 1摘自*Learning and Leading with Habits of Mind*，Arthur L. Costa、Bena Kallick编（ACSD）。

让我们换个角度来看这个问题，在传统教育体系中，我们关注的是答案。但我们更应该关注，或者至少应该花更大力气去强调的是学生在面对未知问题时该怎么办。我们需要具备能帮助找出答案的素质。

说到这里，你是否想过，为什么人们不知道某件事时，不直接说“不知道”？这种现象，可以被部分解释为“达克效应”，无知比知识更容易招致自信。

基本上，在某件事上不擅长的人，不知道他们在这件事上有多无知，因此容易高估自己，而那些擅长某事的人，却因为了解，反而低估了他们的能力。知识削弱了人们的自信。将这些谨记在心，不要高估，也不要低估自己，确保说到的东西可以做到，和其他数据科学家交谈时不要忘记随时检查自己。

> 思维实验：如何教授数据科学
>
> 换作你，如何设计一门数据科学课程，将重点放在思考习惯，而不是技术的培养上？你如何量化它？又如何评价它？哪些内容学生们可以写在自己的简历上？

16.3.3 成为提问者

人们很容易将模型过拟合，这是人们的天性，每个人都“望子成龙”，很可能你已经在这个模型上工作几个月了，对待它，你会像个慈母或慈父。

人们的另一个天性是低估了坏消息，并且将坏消息归罪于别人。站在父母的角度，自己孩子干的事，或能干的事都不会是坏事。除非是有

人使坏，唆使孩子干的。我们该如何克服人类的这种天性？

理想情况下，我们希望数据科学家配得上“科学家”这个称号，他们应该对假设进行检验，不惧挑战，欢迎其他理论进行争鸣。这就是说，我们得挑自己的刺，接受挑战，像个科学家那样设计实验，而不是用花言巧语或以政治为由为自己的模型进行辩护。如果有人说他们能做得更好，那就事先确定好评价标准，让他们去试好了。尽量让事情变得客观。

习惯遵循一份包含关键步骤的标准清单：非要这样做不可吗？如何衡量它？哪种算法适合这个问题，为什么？我该如何评价它？我真的具备做这件事的技术吗？如果没有，我该如何学习这些技术？我可以和谁一起工作？有问题可以问谁？对现实世界有何影响？最后这个可能是最重要的一个问题。

其次，习惯向别人提问。当你遇到一个问题或某个人，需要提出问题时，先假设自己是聪明的，不要认为回答的人比你知道得多或者少。不要试图去证明什么，你的目的是找到答案。像个孩子那样充满好奇心，别怕自己看起来很笨。要求澄清一些符号、术语和流程：数据从哪里来？如何使用这些数据？为什么这些数据能用？我们忽略了哪些数据，这些数据包含更多特征吗？谁该干什么？如何一起合作？

最后，有一个特别重要的概念要时刻牢记，那就是因果关系和相关关系。不要将二者混为一谈。也就是说，当你看到相关关系时，不要误以为那是因果关系。



下一代数据科学家会怎么做

下一代数据科学家仍然对一切保持怀疑的态度：怀疑模型本身，模型会在什么情况下失败，模型该如何使用，模型又会被如何误用。下一代数据科学家知道他们正在构建模型的影响和后果，他们会思考基于反馈的循环和潜在的在模型之间进行博弈。

16.4　做一个有道德感的数据科学家

数据科学家绝不是一个坐在墙角的书呆子，当你工作时，会有越来越

多的伦理问题需要考虑。

现在我们拥有海量的市场和用户行为数据。作为数据科学家，我们不只是机械地使用一些机器学习工具，我们还需要从人文主义角度，去解释和发现数据中的意义，去做出符合道德规范的、基于数据的决策。

要牢记用户行为产生的数据已经构成了数据产品的一部分，反过来数据产品又被用户使用，影响用户的行为。这种例子俯拾皆是：推荐系统、排名算法、好友推荐系统等。这种现象将会在越来越多的行业看到，比如教育、金融、零售、健康等领域。这种基于反馈的循环也会出错，金融风暴就是给我们的一个警示。

数据科学被用来预测未来（Nate Silver，http://slate.me/1g3Di1Y）、预测现在（Hal Varian，http://googleresearch.blogspot.com/2009/04/predicting-present-with-google-trends.html）、探索数据中存在的因果关系（Sinan Aral，http://caossnyc.org/#schedule），这样的例子我们已经讲了很多。

下面我们要讲的是模型和算法不仅用来预测未来，它们还在影响未来。有时，这是我们所期待的，有时，这又是我们想极力避免的。

Emanuel Derman提出了为金融行业进行建模的“希波克拉底誓言”，这份誓言不仅适用于金融业，也适用于其他行业，让我们以此为起点，介绍一下建模时需要做出的道德考量。

- 时刻谨记这个世界不是由我创造的，它不必符合我的方程式。
- 虽然可以大胆地使用模型做预测，但切不可过分迷信数学。
- 我绝不会牺牲真实换来优雅的数学模型，并且拒绝解释这样做的原因。我也不会在模型的准确度上欺骗我的用户，相反的，我会如实陈述模型的假设条件和模型的局限性。
- 我知道我的工作对于社会和经济影响巨大，其中很多已经超出我的理解能力。

这份誓言并没有将在业界工作时的政治因素考虑进来，但作为一个数据科学家，有时不得不如此。即使对模型心存怀疑，仍然会有人置你的警告于不顾，错误地使用模型。所以“建模者的希波克拉底誓言”的约束力在现实中还是显得有点捉襟见肘，但不可否认，这仍然是一个好的开始。



下一代数据科学家会怎么做

下一代数据科学家不会让金钱蒙蔽了双眼，不会将自己的模型用于危害社会的活动。他们寻找机会去解决那些对社会有价值的问题，并且试图了解他们的模型对社会的影响。

最后，有一些使用数据科学服务社会的途径：比如可以作为志愿者参与DataKind（http://www.datakind.org/）的一些长期项目，这可比那些在周末举行的黑客马拉松有意思多了。

还有让社会变得公正透明的途径：Victoria Stodden创办的RunMyCode网站（http://www.runmycode.org/CompanionSite/），旨在让科研工作开源化和可复制化。

我们想暂且把发言权交给来自哥伦比亚大学历史系的Matthew Jones教授，他是历史方面的专家，他也上了我们的数据科学导论课程，他将自己对该课程的一些想法写下来，着重阐述了道德和自大情绪的克服在数据科学中的重要作用。我们把他的文章抄录在此，以飨读者。

数据和自大狂

在2012年美国总统大选后，一种幸灾乐祸的情绪在数据科学家以及那些崇拜甚至神化他们的粉丝中迅速蔓延，因传统的专家们预测错误了。计算统计和数据分析彻底击败了传统的预测方式，这些预测一般是基于旧式直觉、长期的新闻业从业经验，或者是依靠日渐式微的华盛顿内部人士关系网。奥巴马团队和其他人基于量化的预测（http://ti.me/GzJlhH）的成功（http://lat.ms/1hkOxki），显而易见（http://ti.me/GzJlhH）地揭示了政治分析领域一个新时代的到来。传统的所谓“专家意见”，现在鲜被提及，而且有人建议应被请下神坛，让位于新出现的数据驱动的政治分析。

这是一段引人入胜的传奇，充分展示了新旧两种知识在面对同一个问题时的冲突。然而，那些真正优秀的数据科学家已经开始反思，完全抛弃现有的领域知识

和专家们是相当危险的。

关于起源的故事总会为专业知识的分层增加更多的合理性。数据挖掘领域长期以来有一个广为传颂的故事，尽管有点虚构的成分。故事是这样的：利用一种关联算法（https://en.wikipedia.org/wiki/Association_rule_learning），研究者意外地发现，在百货商店里购买尿布的男士常常会同时购买啤酒（http://www.itbusiness.ca/news/behind-the-beer-and-diapers-data-mining-legend/136）。其实，市场营销人员凭借他们自学的有限心理学知识和对市场的直觉，早在计算机还被称作"电脑"之前就发现了这一现象。这个故事符合经典的模板，概率论和统计学从欧洲启蒙运动伊始（http://press.princeton.edu/titles/4295.html）就一直在挑战传统知识：保险和养老年金的价格取决于数据，而不是申请者的状况和那些年长的专家们的建议。在将让人喜爱的（或者让人恐怖的）艾普西隆、德耳塔引入真正的分析那本书中（http://goo.gl/v5JhxG），奥古斯丁·路易·柯西批评了那些法国大革命中的统计学家："让我们满怀热情的去发展数学，但不要妄图将其应用于其他领域；我们不要幻想用公式来攻击历史，也不要通过代数理论和微积分来评判道德的高下。"

这些描述与当年兴起的分离主义理念相当契合，而这种理念正是硅谷的自由主义者、信奉熊彼特理论的资本主义者和一些技术期刊的核心主张。虽然可避免政治分析中的权力寻租和其他规则，但二分法错误地分开了技能和知识，而这两者却都是牵引数据科学不断进步的关键。前面的章节主要讲述了培养数据科学家掌握多种技能的各种方法——这将粗浅的二分法观点驳斥得体无完肤，也就是说，数据专家和传统的专家并非毫无交集。本书在教授技术的同时，让那些自大狂变得谦逊，尤其是那些自负算法无敌的人。

奥巴马的数据团队这样解释他们的成功（http://goo.gl/sB8pGH），成功来自严肃地对待自负情绪，构建的技术系统避免了过分估计，选择了将算法作为后台和网络的补充。Haper Reed向亚特兰大报的作者Alexis Madrigal这样解释道："我认为共和党搞砸

了。我知道我们有最好的团队，但是我们不知道它是否能工作。我曾经信誓旦旦这一定能行，我把身家都压在了上面。我们有时间，有资源，我们做了所有能做的，但还是不起作用。总有些事情会发生的。”

有关“领域知识”价值的讨论在数据科学社区里长期呈现两极分化的趋势。无监督学习其实是克服了对人们习惯的社会和科学分析方式的依赖，正如在奥巴马分析团队中看到的那样（http://lat.ms/1hkOxki）。年仅29岁的首席分析官Daniel Wagner这样说：

> 在竞选活动中找出“足球妈妈”“服务员妈妈”分组游说这样的方式已经过时了。竞选活动现在已经可以精确定位到每个中间选民。郊区的白人妇女？他们都是不一样的。拉丁裔社区差异较大，不同人有不同的兴趣。数据能给你的只是如何区分出这些差异。
>
> 人们渐渐弱化了分组，但是将领域知识带入统计学的运动似乎和正规的数据挖掘一样历史悠久。

《华尔街日报》有篇文章（http://on.wsj.com/15LnZno），现在已经不那么有名了。Peggy Noonan这样形容奥巴马分析团队的工作：”这就像火星人干的。“竞选活动中参与的人很少，作战室里全是些“高科技不会流血”的物种。与此同时罗姆尼一方的广告也类似。

数据科学基于算法但不等同于算法。算法的使用要基于社会学中的“隐性知识”（http://amzn.to/19huR1W）——这是通过实践得来的知识，不易简单总结成规则，或者根本就不可能总结成规则。使用好算法从根本上说也是一项人为活动——一种非算法的东西。

不去警告这些年轻的数据科学家就会带来很多过拟合的危险，在训练集上采用了噪声数据，或者过多学习训练集以致不能将结论泛化。避免过拟合需要仔细考

虑使用的算法。算法是需要我们投入更多思考的工具。Peter Huber在1977年这样解释道："我认为，问题不是要用机器智能取代人类的聪明才智，而是使用所有计算机科学、人工智能提供的工具帮助人类发挥自己的聪明才智，尤其是即兴使用各种搜索工具、记录分析的进展。""即兴"一词恰好指出了要掌握工具的使用、依据上下文推理、避免死记硬背。自大狂使用算法时，务必要深入理解算法，对具体的实现要了如指掌。

Noonan提供的奥巴马竞选团队的招聘广告上（http://goo.gl/3KuIuj），明显体现出对现有模型优点和缺点的思考：

- 开发和实现统计/预测/机器学习模型支持竞选活动、数字媒体、付费媒体和募捐竞选资金；
- 评价以前模型的性能，决定是否需要更新；
- 设计和执行实验，验证模型的适用性和有效性。

自动化的模型唾手可得，并不是这里的重点：重点是批评和评价。只有熟悉这一领域的火星人才能干这个：各种各样的数据太重要了，千万不可放过。

怎么样学会随机应变？换句话说，什么模型是教育数据科学家的最佳选择？能够使用算法和大数据来即兴发挥的能力需要持续不断地抽丝剥茧，清理组织混乱、不完整、很可能是没有结论的数据。训练需要的最佳模型不是通过这种狭隘的职业教育，而是回归人文艺术的本质。

几个世纪以来，艺术，比如数学和音乐都被归为人文学科的范畴，因为它们不能被自动化、机器化、重复化和惯常化。人文学科让人得到自由，这种自由反映在他们使用的工具、行为和习惯上。人们不被工具所奴役，他们可以自由地选择使用或不使用工具。算法也是如此，称职的数据科学家不必循规蹈矩，任何技术相关的宿命论都不适用。数据科学家从不将使用一项技术的可能性和必要性混为一谈。面对数据有无数可能，但是只有很小的一部分合乎道德（并且有趣）的问题需要我们花力气去解决。

— Matthew Jones

16.5 对于职业生涯的建议

对于雄心勃勃的下一代据科学家，尤其那些已经阅读至这里的读者朋友们，我们是从来不吝惜自己的建议的。

我们已经习惯回答这类问题了，毕竟，很多人都曾问过我们他们是否应该成为一名数据科学家。相比直接给出答案，我们经常会先问他们两个问题。

1. 你选择什么样的生活

要回答这个问题，你需要清楚自己看重什么。你也许看重金钱，你需要足够的钱过上你理想的生活，甚至想要更多的钱。这样很多很酷、但是没人愿意掏钱的项目就被排除在你的选择之外了（但千万不要因此放弃为这类项目筹款）。你也许看重和爱人或者朋友共度美好时光，那么你就不应该选择去一些初创公司工作，在那里的人一天工作十二小时，而且经常睡在办公桌底下。我没骗你，这样的地方的确存在。

或者你追求的是为世界干点有益的事，同时实现自我价值并体现智慧价值。一定要根据个人情况去衡量这些选择，它们是截然不同的选择。

你的目标是什么？你想追求什么？你想成名吗？你想受人尊敬，并且学有所长吗？或许你的最佳选择是上述几种情况的某种组合，那么现在，你清楚地知道你的选择了吗？

2. 你有哪些局限

有很多外在因素，或许是你无法左右的，比如你无法选择和家人住在哪里。还有金钱和时间上的限制，你是否需要研究公司的年假、产假和陪产假政策。向用人单位推销自己难度几何？不要被逼入绝境，想想如何展示自己积极的一面：学历、优缺点、能改变的和不能改变的。

基于你看重的东西和那些局限因素，有很多方案可供选择。在我们看来，工作没有好坏，只有合适不合适。不同的人想从工作中获得的东西是不一样的。

一方面，你决定要做的任何事都不是绝对的，你随时可以改变当初的选择，人们不都在换工作吗？因此不要太过担心。另一方面，人生苦短，不要停滞不前，永远向着正确的方向前进，永远追寻那些内心真正在乎的东西。

最后，如果你认为你的想法和思考方式和周围人不同，那么请相信自己，去探索它，你有可能大有作为。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_interview，讲述码农精彩人生
- 微信 图灵教育：turingbooks

091507240605ToBeReplacedWithUserId

版权信息

书名：SQL反模式

作者：Bill Karwin

译者：push-chen， 谭振林

ISBN：978-7-115-26127-4

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

您购买的图灵电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

目录

版 权 声 明
读 者 感 言
译 者 序 一
译 者 序 二
第1章　引言
第一部分　逻辑型数据库设计反模式
　第2章　乱穿马路
　第3章　单纯的树
　第4章　需要ID
　第5章　不用钥匙的入口
　第6章　实体—属性—值
　第7章　多态关联
　第8章　多列属性
　第9章　元数据分裂
第二部分　物理数据库设计反模式
　第10章　取整错误
　第11章　每日新花样
　第12章　幽灵文件
　第13章　乱用索引
第三部分　查询反模式
　第14章　对未知的恐惧
　第15章　模棱两可的分组
　第16章　随机选择
　第17章　可怜人的搜索引擎
　第18章　意大利面条式查询
　第19章　隐式的列
第四部分　应用程序开发反模式
　第20章　明文密码
　第21章　SQL注入
　第22章　伪键洁癖
　第23章　非礼勿视
　第24章　外交豁免权
　第25章　魔豆
第五部分　附录
　附录A　规范化规则
　附录B　参考书目

版 权 声 明

Copyright © 2010 Bill Karwin. Original English language edition, entitled *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*.

Simplified Chinese-language edition copyright ©2011 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由The Pragmatic Programmers, LLC.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

读者感言

对于经常碰到本书中表述的那些数据库设计决择的软件开发人员来说，这本书是必读的。因为它将帮助开发团队理解数据库设计达成的结果，并且基于实际的需求、预期、测定做出最合理的决定。

——Darby Felton，DevBots Software Development的联合创始人

我非常喜欢Bill在书中采用的写作方式，展示了他独一无二的风格和幽默感，这对讨论一大堆枯燥的话题太重要了。Bill成功运用了一种很好的表述方式，让技术以易于理解的面貌示人，而且便于以后查阅。简而言之，这将是你书架里又一极好的资源。

——Arjen Lentz，Open Query

（**http://openquery.com**）执行总监，

High Performance MySQL第二版作者之一

对于有SQL基础，但是在为项目设计SQL数据库时希望寻求基础之外帮助的软件工程师来说，这本书尤其有用。

——Liz Neely，资深数据库程序员

Bill捕捉到了我们在SQL不同方面碰到过的很多陷阱的关键所在，而我们有些时候甚至没 有意识到已被困住了。Bill的反模式涵盖从“不敢相信我又犯了一遍”的事后感叹，到最佳方案与伴你一路走来的SQL教条相左的诡异情况。这是一本对SQL骨灰和新手都不错的书。

——Danny Thorpe，

Microsoft总工程师；

Delphi Component Design作者

译者序一

毫无疑问，数据库领域当下最热门的概念是NoSQL，我正在公司最新大型社区项目中实践NoSQL产品，并准备在更大范围内推广优秀的NoSQL产品，而另一译者——陈魏明，则自己研发了一个NoSQL产品，应用在另一大型社区项目中，提供Feed系统的支持。

但是，正如NoSQL自身所宣扬的一样，任何一种NoSQL产品，甚至所有的NoSQL产品合在一起，它们的设计初衷绝不是解决掉所有的数据处理需求，它们追求的是为某一种或某几种数据处理场景选择最优的CAP[1]组合，提供最合适的解决方案。

> [1] CAP是指：一致性（Consistency），可用性（Availability），分区容忍性（Partition tolerance）。CAP原理认为这三个要素最多只能同时实现两点，不可能三者兼顾。

因此，SQL并没有因为NoSQL的流行而变得不重要，它仍然跟以前一样重要，并且因为它长期以来在开发人员中建立的深厚基础，以及丰富的支持工具，特别是强大的查询功能，将使其长期在广泛的数据处理场景中作为主要的解决方案而存在。比如你总不能用Redis[2]来处理运营团队天天变着戏法要的运营分析数据吧。

> [2] Redis是一款优秀的、基于内存处理数据的、原生支持多种数据结构的Key-Value型NoSQL产品。

这本语言略显啰嗦的书，是一本非常实用的书，因为它每一章的内容都源自于最常见、最普通的SQL应用场景，每一章中描述的问题，都是全世界的SQL应用人员犯得最多的错误。总之，我译完这本书后，就有一个强烈的感触：“原来我犯了这么多错误！”

所以，这本书中的知识与教训，应该对很多人（比我资浅的那一小部分人和比我资深的那一大部分人）都有帮助，而且长期有效。

本书作者知识渊博，原作中不少地方引经据典，我们在翻译过程中尽量通过Google等办法查找对应的典故，不过文化的差异和有限的水平，造成译稿中必定还有不少不尽如人意的地方，请各位读者原谅。

谭振林

2011年5月

译者序二

我本人并非DBA，但工作中时刻都要和数据打交道，无论是SQL还是NoSQL。在一个产品的生命周期里，设计与开发总是只占很小的一部分，大量的时间都用在后续的维护、优化和调整中。互联网服务更是如此。而维护、性能调优阶段，就是一个不断地犯错—纠正—归纳的循环。本书作者将他所归纳整理出的这些“错误”写在了这本书中，让我们这些后来者能够更早地发现程序中的问题，从而节省大量的时间成本。

无论时下NoSQL技术有多么的火爆，终究有一个无法解决的问题——内存开销。因而几乎所有的网站、社区，其后台必然有大量使用SQL进行存储的模块。

在翻译本书的过程中，我也回顾了以前所接触到的或者自己做的数据库设计，很多都没有从数据库本身出发去解决问题，而是直接在前端增加了一个缓存。

本书中的很多问题解决方案都很优雅，在不触碰程序员追求程序整洁度的洁癖神经的情况下，利用SQL本身就有的特性和功能解决了问题。这些案例非常值得我们学习。

本书作者知识丰富，在很多方面上都有所涉及，书中提到很多文化方面的内容，翻译不到位之处，敬请指正。

Push Chen

2011年5月

第1章　引言

> 所谓专家，就是在一个很小的领域里把所有错误都犯过了的人。
>
> 尼尔斯·玻尔

我曾经拒绝过第一份关于SQL的工作。

获得加州大学计算机与信息科学的本科学位后不久，我接到了一个工作邀请，来自于一位曾经在加州大学工作过的经理，我们在一次校园活动相识。他当时刚刚建立了自己的软件公司，致力于使用shell脚本和诸如awk的相关工具（诸如Ruby、Python、PHP，甚至Perl等现代动态语言，在当时都还不甚流行），来开发适用于众多UNIX平台的数据库管理系统。这个经理邀请我是因为他需要一个人来开发一些代码，识别并且执行功能有限版本的SQL语言。

他说："我不需要支持完整的SQL语言，那样工作量太大了。我只需要支持一个SQL语句：`SELECT`。"

我在学校里并没有学过SQL。数据库在当时并不像现在这样普遍，并且当时也没有诸如MySQL和PostgreSQL之类的开源程序。但我用shell脚本开发过完整的应用程序，并且了解一些语法分析的技术，也做过一些关于编译器设计和计算语言学的课程项目。因此我在想是否要接受这个工作。只解析像SQL这样的专业语言中单独的一类语句会有多困难呢？

我找了一份SQL的参考资料并且立刻意识到它不同于那些支持`if()`、`while()`语句、变量定义、表达式以及可能还有函数调用的语言。说`SELECT`只是此类语言中的一个语句，等同于说引擎只是汽车的一部分。从字面上来看，这两句话都是正确的，但是都完全掩盖了这两个主体的复杂性和深度。我意识到仅仅为了支持`SELECT`这一个语句的执行，就必须要实现一个完整的关系型数据库管理系统以及其查询引擎的全部代码。

我拒绝了这个用shell脚本实现SQL解析与关系型数据库管理系统引擎的工作机会。那个经理并没有充分理解他的项目的复杂度，可能他并不理解什么是关系型数据库管理系统（RDBMS）。

我早期使用SQL的经历和普通的软件开发人员并没有什么区别，和那些从计算机科学专业毕业的学生相比也是如此。大多数人学习SQL语言都是因为项目所需而不得不自学的，而不是像其他的编程语言那样从头开始认真地学习。不论是对SQL爱好者、专业程序员或者拥有博士学位的娴熟的研究人员，SQL就好像是程序员的一个不经过训练就学会了的软件技能。

此后，当我了解了一些SQL方面的知识后，我惊讶于它和那些过程式的编程语言（诸如C、Pascal和shell）或是面向对象的编程语言（诸如C＋＋、Java、Ruby或Python）都有着显著的区别。SQL是一门声明式的编程语言，就像Lisp、Haskell或者XSLT。SQL使用集合（set）作为根本的数据结构，而面向对象的语言使用的是对象（object）。受过传统培训的软件开发人员被所谓的“阻抗失配”[1]所阻碍，因此很多程序员转而使用现成的面向对象的库，以此来避免学习如何高效地使用SQL。

> [1] 阻抗失配（impedance mismatch）原意为当滤波器输出阻抗Z0和与之端接的负载阻抗ZL不相等时，在该端口A上产生反射，引入到计算机科学中指面向对象应用程序向关系型数据库存储数据时的数据不一致问题。——译者注

自1992年以来，我在工作中大量使用SQL。我在开发应用程序的过程中要使用它，我也为InterBase RDBMS产品提供技术支持、开发培训课程以及文档，并且开发了Perl和PHP中用于SQL编程的库。我在那些网络邮件列表和新闻讨论组中回答了成千上万的问题。大量重复的问题表明程序员总是一遍又一遍地犯同样的错误。

1.1　谁需要这本书

不管你是初学者还是资深人员，这本《SQL反模式》[2]都能帮助需要使用SQL的程序员更有效地使用它。我和所有不同经验层次的人交流过，他们都能从这本书中获益。

> [2] 反模式英文为Antipattern，在SQL中，pattern有时译为范式，本书中采用模式一词。——译者注

你可能已经阅读过SQL语法的参考资料，知道所有的`SELECT`语句的子句，并且能够用它来做一些事情了。渐渐地，你可以通过阅读别人的程序代码或者文章来提升你的SQL技能。但你怎么能区分优劣？怎

么能确定你正在学习的是最佳方法，而不是另一个可能使你陷入困境的方法？

你可能会在本书中找到一些熟悉的话题。即使你之前已经找到了解决方案，仍可以发现新的看待问题的方法。重新审视那些广为流传的错误，将能更好地确认和巩固你对优秀范例的理解。还有一些话题可能对你来说比较新鲜，我希望你能通过阅读它们来改善自己的SQL编程习惯。

如果你是一个训练有素的数据库管理员，可能已经知道了如何用最好的方法来避免本书所描述的SQL编程中易犯的错误。然而这本书也能从软件开发人员的角度来帮助你。开发人员和DBA之间为项目争论是很常见的，但相互尊重和团队合作能够帮助我们更有效地工作。你可以使用本书来向你负责开发的同事解释一些好的做法，以及不这么做会有怎样的后果。

1.2　本书内容

什么是“反模式”？反模式是一种试图解决问题的方法，但通常会同时引发别的问题。反模式虽以不同的形式被广泛实践，但这其中仍存在一定的共通性。人们可能独立地，或是在一个同事、一本书或者一篇文章的帮助下想出一个反模式的主意。很多面向对象的软件设计与项目管理方面的反模式都记录在Portland Pattern Repository[3]中，或是记录在1998年出版的《反模式》[4]（William J. Brown等）一书中。

> [3] Portland Pattern Repository：http://c2.com/cgi-bin/wiki?AntiPattern。
>
> [4] 本书由人民邮电出版社于2007年出版。——编者注

本书描述了我做技术支持、做培训课程、和程序员一起开发软件以及在网络论坛上回答问题时遇到的最常见的错误。这其中很多错误我自己也犯过，除了在深夜花好几个小时找到并解决掉之外并没有更好的办法。

1.2.1　本书结构

这本书将反模式分类成如下四部分。

逻辑数据库设计反模式

在你开始编码之前，需要决定数据库里存储什么信息以及最佳的数据组织方式和内在关联方式。这包含了如何设计数据库的表、字段和关系。

物理数据库设计反模式

在知道了需要存储哪些数据之后，使用你所知的RDBMS技术特性尽可能高效地实现数据管理。这包含了定义表和索引，以及选择数据类型。你也要使用SQL的“数据定义语言”，比如 `CREATE TABLE`语句。

查询反模式

你需要向数据库中添加然后获取数据。SQL的查询是使用“数据操作语言”来完成，比如`SELECT`、`UPDATE`和`DELETE`语句。

应用程序开发反模式

SQL应该会用在使用C＋＋、Java、Php、Python或者Ruby等语言构建的应用程序中。在应用程序中使用SQL的方式有好有坏，该部分内容描述了一些常见错误。

很多反模式章节都采用了一些比较幽默或是能产生共鸣的标题，比如金锤子、重新造轮子或由委员会设计。通常来说，会同时给出正面的设计模式和反模式的名字作为隐喻或帮助记忆。

附录提供了一些对关系数据库理论实践的描述。本书中的很多反模式其实都是对数据库理论的误解造成的。

1.2.2　反模式分解

每个反模式章节都包含了如下的子标题结构。

目的

这是你可能要去尝试解决的任务。意图使用反模式提供解决方案，但通常会以引起更多问题 而告终。

反模式

这一部分表述了通常使用的解决方案的本质，并且展示了那些没有预知到的后果，正是这些使得这些方案成为反模式。

如何识别反模式

一些固定的方式会有助于你辨识在项目中使用的反模式。你遇到的特殊障碍，或是你自己和别人说的一些话，都能使你提前识别出反模式。

合理使用反模式

规则总有例外。在某些情况下，本来认为是反模式的设计却可能是合理的，或者说至少是所有的方案中最合理的。

解决方案

这一部分描述了首选的解决方案，它们不仅能够解决原有的问题，同时也不至于引起由反模式导致的新问题。

1.3　本书未涉及的内容

我不打算讲解SQL的语法或者相关术语。有大量关于这些基础内容的书籍或者网络资料。我假设你已经学习了足够多的SQL的语法来使用这门语言并且能够做好一些事情。

性能、可伸缩性以及优化对于很多开发数据驱动应用程序，特别是网络应用的设计人员来说是非常重要的。市面上也有很多和数据库开发性能相关的书籍。我推荐*SQL Performance Tuning*[GP-3]和*High Performance MySQL, Second Edition*[SZT＋08][5]。本书的一些主题和性能相关，但这不是本书最主要的目的。

> 5《高性能MySQL（第二版）》由电子工业出版社于2010年出版。——编者注

我尝试把问题表述成适用于所有厂商的数据库，并且这些解决方案也将会适用于所有的数据库。SQL语言被规定为一种ANSI和ISO标准，所有的数据库都支持这一标准，因此我尽可能中立地使用SQL而不偏向任何品牌，并且我会明确说明不同厂商SQL数据库的特定扩展。

数据访问框架和对象关系映射（ORM）库是非常有用的工具，但这些也并不是本书的重点。我用最普通、最直白的方式写过很多PHP的代码范例。本书的范例足够简单，从而能够在大部分编程语言中适用。

数据库的管理和操作任务，诸如服务器磁盘分配、安装和配置、监控

和备份、日志分析以及安全都是非常重要并且值得用一整本书来描述的，但本书更倾向于那些使用SQL的开发人员而不是数据库管理员。

这本书是关于SQL和关系型数据库的，以及其他的替代技术，诸如面向对象数据库、键/值存储、面向列的数据库、面向文档的数据库、分级数据库、网络数据库、Map/Reduce框架或者语义数据存储。比较这些技术的优缺点并在不同的数据管理解决方案中恰当地使用它们是一个 很有趣的课题，但这并不是本书的议题。

1.4　规约

下面描述了本书中使用的一些规约。

排版

SQL关键字都大写并且使用等宽字体，以使得它们在上下文中更突出，就像`SELECT`。

SQL的表名，也用等宽字体，并且首字母大写，如`Accounts`或者`BugsProducts`。SQL的列，也使用等宽字体，全都使用小写，并且使用下划线分词，如`account_name`。

术语

SQL的正确发音是S-Q-L，不是C-QL。虽然我对通俗用法并没有什么反对意见，但我更倾向于使用正式用法。

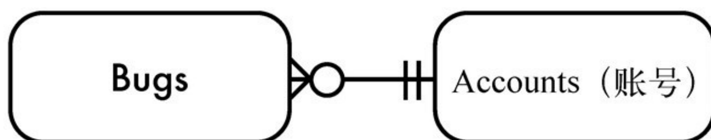
在数据库相关的用法中，“索引”一词指的是一个有序的信息集合。在其他情况下“索引”可能指的是指示器。

在SQL中，术语查询（query）和语句（statement）有时指的是同一个意思，指的都是任何可执行的一句完整的SQL指令。为了描述清晰，我使用“查询”表示`SELECT`语句，而“语句”用来表达其他语句，包括`INSERT`、`UPDATE`和`DELETE`以及数据定义语句。

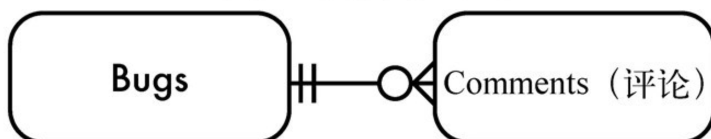
数据实体关系图

最常见的关系数据库图表就是实体关系图ERD（Entity Relationship Diagram）。表格使用矩形表示，关系使用链接各个矩形的线段来表示，并且在两端使用各种符号来表述关系的基数。具体事例可以参考下一页的图1-1。

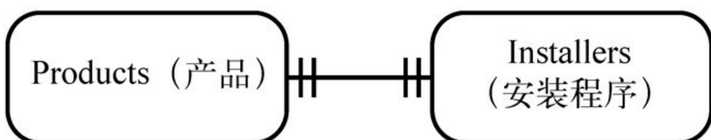
多对一
每个账号记录了很多 bug



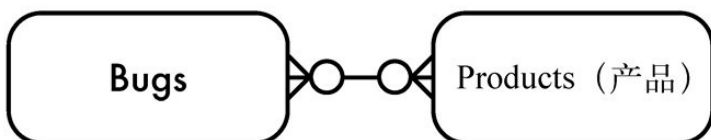
一对多
每个 bug 有很多评论



一对一
每一个产品对应一个安装程序



多对多
每个产品可能有很多 bug；
一个 bug 可能属于多个产品



多对多
同上，交叉表



图1-1　实体关系图ERD示例

1.5 范例数据库

我使用一个假想的缺陷跟踪程序的数据库来展示大部分与SQL反模式相关的话题。图1-2是该数据库的ERD。请注意，Bugs表和Accounts表之间有3个连接，代表3个不同的外键。

接下来的数据库定义语句则展示出如何定义这些表。有些时候所做的选择只是为了照顾后面的范例，因而它们可能并不是人们在实际应用程序中所做的选择。我尽力使用标准SQL来定义这个数据库，使其能适用于任一数据库产品，但也会出现一些MySQL数据类型，诸如SERIAL和BIGINT语句。

Introduction/setup.sql

```
CREATE TABLE Accounts (
  account_id         SERIAL PRIMARY KEY,
  account_name       VARCHAR(20),
  first_name         VARCHAR(20),
  last_name          VARCHAR(20),
  email              VARCHAR(100),
  password_hash      CHAR(64),
  portrait_image     BLOB,
  hourly_rate        NUMERIC(9,2)
);

CREATE TABLE BugStatus (
  status             VARCHAR(20) PRIMARY KEY
);

CREATE TABLE Bugs (
  bug_id             SERIAL PRIMARY KEY,
  date_reported      DATE NOT NULL,
  summary            VARCHAR(80),
  description        VARCHAR(1000),
  resolution         VARCHAR(1000),
  reported_by        BIGINT UNSIGNED NOT NULL,
  assigned_to        BIGINT UNSIGNED,
  verified_by        BIGINT UNSIGNED,
  status             VARCHAR(20) NOT NULL DEFAULT 'NEW',
  priority           VARCHAR(20),
  hours              NUMERIC(9,2),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
```

```
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);

CREATE TABLE Comments (
  comment_id        SERIAL PRIMARY KEY,
  bug_id            BIGINT UNSIGNED NOT NULL,
  author            BIGINT UNSIGNED NOT NULL,
  comment_date      DATETIME NOT NULL,
  comment           TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);

CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
  image_id          BIGINT UNSIGNED NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY      (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

CREATE TABLE Tags (
  bug_id            BIGINT UNSIGNED NOT NULL,
  tag               VARCHAR(20) NOT NULL,
  PRIMARY KEY      (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

CREATE TABLE Products (
  product_id        SERIAL PRIMARY KEY,
  product_name      VARCHAR(50)
);

CREATE TABLE BugsProducts(
  bug_id            BIGINT UNSIGNED NOT NULL,
  product_id        BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY      (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

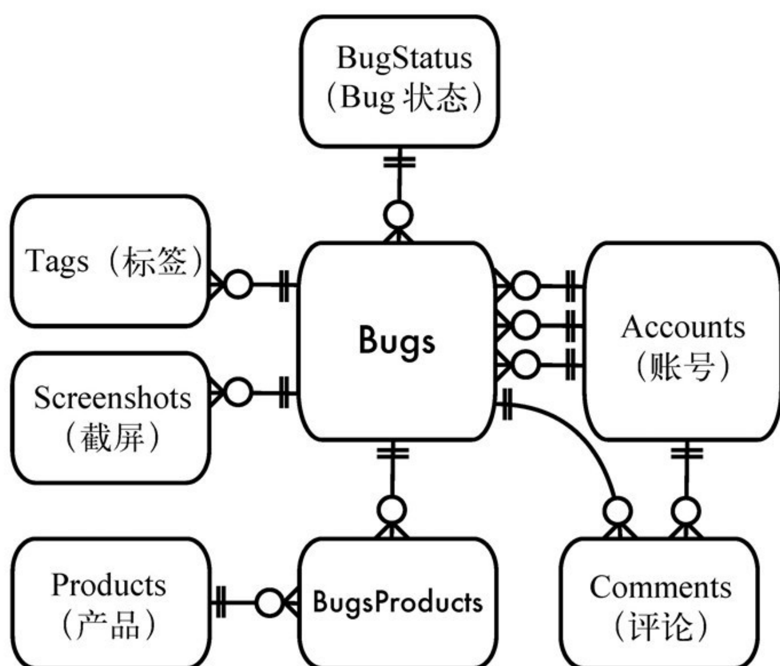


图1-2　BUG数据库ERD图

在一些章节中，尤其是在逻辑数据库反模式的章节中，我展示了一些不同的数据库定义，既为了展示反模式，也同时展示一个可选的解决方案来避免反模式。

1.6　致谢

首先，我要感谢我的妻子Jan，没有她的启发、关爱和支持，更不用说时时的督促，我无法写成这本书。

我也想要对我的审阅者表达谢意，感谢他们花了那么多的时间。他们的建议提高了本书的质量。他们是Marcus Adams、Jeff Bean、Frederic Daoud、Darby Felton、Arjen Lentz、Andy Lester、 Chris Levesque、Mike Naberezny、Liz Nealy、Daev Roehr、Marco Romanini、Maik Schmidt、Gale Straney以及Danny Thorpe。

感谢编辑Jacquelyn Carter和Pragmatic Bookshelf的出版人，他们相信本书一定不辱使命。

第一部分　逻辑型数据库设计反模式

本部分内容

- 第2章　乱穿马路
- 第3章　单纯的树
- 第4章　需要ID
- 第5章　不用钥匙的入口
- 第6章　实体—属性—值
- 第7章　多态关联
- 第8章　多列属性
- 第9章　元数据分裂

第2章　乱穿马路

> **8一个不愿透露姓名的Netscape工程师曾经将一个指针的地址当成字符串传给了JavaScript，随后再回传给C，节省了30秒。**
>
> 布雷克·罗斯

假设你正在为缺陷跟踪程序开发一种功能，将某个用户指定为某个产品的主要联系人。你最初的设计只允许每个产品拥有一个联系人，然而，就像你猜的那样，需求可能会变为支持“每个产品有多个联系人”。

此时，将数据库中原来存储单一用户标识的字段改成使用逗号分隔的用户标识列表，似乎是一个很简单且合理的解决方案。

很快，你的老板跑来问你：“工程部在往他们的项目中添加相关人员时发现最多只能添加5个人，如果想要继续添加更多的人，程序就会报错，这是怎么回事？”

你点头说道：“是的，只能在一个项目中列出这么多人，就是这么设计的。”你觉得这是一件再正常不过的事情了。

但老板似乎并不这么认为，他需要一个更加明确的解释。“好吧，5到10个，可能再多存几个，那取决于这些账号创建时间的早晚。”老板感觉非常吃惊，于是你继续说道：“我使用逗号分隔的列表来存储项目的账号ID，而这个ID列表的长度不能超过字符串的最大长度值。账号ID越短，列表中的账号ID就越多。因此，账号创建较早的人，他们的ID不大于99，这些账号ID更短。”

老板皱起眉头，你觉得你又要加班到很晚了。

程序员通常使用逗号分隔的列表来避免在多对多的关系中创建交叉表，我将这种设计方式定义为一种反模式，称为乱穿马路（Jaywalking），因为乱穿马路也是避免过十字路口的一种方式。

2.1 目标：存储多值属性

设计一个单值表列是非常简单的：你可以选择一个SQL的内置数据类型，以该类型来存储这个表项的数据，比如整型、日期类型或者字符串。但是如何才能做到在一列中存储一系列相关数据的集合呢？

在我们的缺陷跟踪数据库的例子中，我们在Products表中使用一个整型的列来关联产品和对应的联系人。每个账号可能对应很多产品，每个产品又引用了一个联系人，因此我们在产品和账号之间有一个多对一的关系。

Jaywalking/obj/create.sql

```
CREATE TABLE Products (
  product_id   SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id   BIGINT UNSIGNED,
  -- . . .
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', 12);
```

随着项目日趋成熟，你意识到一个产品可能会有多个联系人。除了多对一的关系之外，我们还需要支持产品到账号的一对多的关系。Products表中的一行数据必须要能够存储多个联系人。

2.2 反模式：格式化的逗号分隔列表

为了将对数据库表结构的改动控制在最小范围内，你决定将account_id的类型修改成VARCHAR，这样就可以列出该列中的多个账号ID，每个账号ID之间用逗号分隔。

Jaywalking/anti/create.sql

```
CREATE TABLE Products (
  product_id   SERIAL PRIMARY KEY,
  product_name VARCHAR(1000),
  account_id   VARCHAR(100), -- comma-separated list
  -- . . .
);
```

这样的设计似乎可行，因为你没有创建额外的表或者列，而仅仅改变了一个字段的数据类 型。然而，我们来看一下这样设计所必须承受的性能问题和数据完整性问题。

2.2.1 查询指定账号的产品

如果所有的外键都合并在一个单元格内，查询会变得异常困难。你将不能再使用等号，相反，不得不对某类模式使用测试。比如，MySQL下可以写一些如下的表达式来查询所有账号ID为12的产品：

Jaywalking/anti/regexp.sql

```
SELECT * FROM Products WHERE account_id REGEXP '[[:<:]]12[[:>
```

模式匹配的表达式可能会返回错误结果，并且无法享受索引带来的性能优势。由于模式匹配表达式的语法在不同品牌的数据库中是不同的，因此你的SQL代码并不是平台中立的。

2.2.2 查询指定产品的账号

同样地，使用逗号分隔的列表来做多表联结查询定位一行数据也是极不优雅和耗时的。

Jaywalking/anti/regexp.sql

```
SELECT * FROM Products AS p JOIN Accounts AS a
    ON p.account_id REGEXP '[[:<:]]' || a.account_id || '[[:>
WHERE p.product_id = 123;
```

联合两张表并使用如上的一句表达式将毁掉任何使用索引的可能。这样的查询必须扫描两张表，创建一个交叉结果集，然后使用正则表达式遍历每一行联合后的数据进行匹配。

2.2.3 执行聚合查询

聚合查询使用SQL内置的聚合函数，如COUNT()、SUM()、AVG()。然而，这些函数是针对分组行而设计的，并不是为了逗号分隔的列表。你不得不借助于如下的一些方法：

Jaywalking/anti/count.sql

```
SELECT product_id, LENGTH(account_id) - LENGTH(REPLACE(accoun
    AS contacts_per_product
FROM Products;
```

这类方法可能看上去很高明，但并不清晰。这种类型的解决方案需要花费很长的时间来开发并且不方便调试。何况有些聚合查询根本无法使用这些技巧来完成。

2.2.4 更新指定产品的账号

你可以用字符串拼接的方式在列表尾端增加一个新的ID，但这并不能使列表按顺序存储。

Jaywalking/anti/update.sql

```
UPDATE Products
SET account_id = account_id || ',' || 56
WHERE product_id = 123;
```

要从列表中删除一个条目，必须执行两条SQL查询语句：第一条提取老的列表，第二条存储更新后的列表。

Jaywalking/anti/remove.php

```
<?php

$stmt = $pdo->query(
  "SELECT account_id FROM Products WHERE product_id = 123");
$row = $stmt->fetch();
$contact_list = $row['account_id'];

// change list in PHP code
$value_to_remove = "34";
$contact_list = split(",", $contact_list);
$key_to_remove = array_search($value_to_remove, $contact_list
unset($contact_list[$key_to_remove]);
$contact_list = join(",", $contact_list);

$stmt = $pdo->prepare(
  "UPDATE Products SET account_id = ?
   WHERE product_id = 123");
$stmt->execute(array($contact_list));
```

仅仅为了从列表中删除一个账号就要写如此多的代码。

2.2.5 验证产品ID

用什么来防止用户在ID中输入诸如“banana”（香蕉）这样的非法字段？

Jaywalking/anti/banana.sql

```
INSERT INTO Products (product_id, product_name, account_id)
VALUES (DEFAULT, 'Visual TurboBuilder', '12,34,banana');
```

用户总能找到办法输入他们想输入的东西，然后你的数据库就会变得很乱。上面这样的情况并不一定是一个数据库错误，但相对应的数据会变得毫无价值。

2.2.6 选择合适的分隔符

如果存储一个字符串列表而不是数字列表，列表中的某些条目可能会包含分隔符。使用逗号作为分隔符可能会有问题。当然，你到时候可以再换一个字符，但你能确保这个新字符永远不会出现在条目中吗？

2.2.7 列表长度限制

你能在一个VARCHAR(30)的结构中存多少数据呢？这依赖于每个条目的长度。如果每个条目只有2个字符长，那你能存10个条目（包括逗号）。但如果每个条目的长度为6，你就只能存4个了。

Jaywalking/anti/length.sql

```
UPDATE Products SET account_id = '10,14,18,22,26,30,34,38,42,
WHERE product_id = 123;

UPDATE Products SET account_id = '101418,222630,343842,467790
WHERE product_id = 123;
```

你怎能确定VARCHAR(30)能够支持你未来所需的最长列表呢？多长才够长？你可以自己尝试着去和老板或者客户解释这么限制的原因。

2.3 如何识别反模式

如果你的项目团队说过下面这些话，那么这很有可能就是在项目中使用了“乱穿马路”设计模式的线索。

- “列表最多支持存放多少数据？”

  这个问题在选择VARCHAR列的最大长度时被提及。

- “你知道在SQL中如何做分词查找吗？”

  如果你用了正则表达式来提取字符串中的组成部分，这可能是一种提示，意味着你应该把这些数据分开存储。

- “哪些字符不会出现在任何一个列表条目中？”

  你想要使用一个不会令人困惑的分割符号，但你也应该明白任何字符都有可能在某天出现在字段中的某个值内。

2.4 合理使用反模式

出于性能优化的考量，可能在数据库的结构中需要使用反规范化的设计。将列表存储为以逗号分隔的字符串就是反规范化的一个实例。

应用程序可能会需要逗号分隔的这种存储格式，也可能没必要获取列表中的单独项。同样，如果应用程序接收的源数据是有逗号分隔的格式，而你只需要存储和使用它们并且不对其做任何修改，完全没必要分开其中的值。

你需要谨慎使用反规范化的数据库设计。尽可能地使用规范化的数据库设计，因为那样的设计能让你的产品代码更灵活，并且也能在数据库层保持数据完整性。

2.5 解决方案：创建一张交叉表

将account_id存储在一张单独的表中，而不是存储在Products表中，从而每个独立的account值都可以占据一行。这张新表称为Contacts，实现了Products和Accounts的多对多关系。

Jaywalking/soln/create.sql

```
CREATE TABLE Contacts (
  product_id  BIGINT UNSIGNED NOT NULL,
  account_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (product_id, account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

INSERT INTO Contacts (product_id, accont_id)
VALUES (123, 12), (123, 34), (345, 23), (567, 12), (567, 34);
```

当一张表有指向另外两张表的外键时，我们称这种表为一张交叉表[1]，它实现了两张表之间的多对多关系。这意味着每个产品都可以通过交叉表和多个账号关联；同样地，一个账号也可以通过交叉表和多个产品关联。参考图2-1。

1 有人用“联合表”、“多对多表”、“映射表”或其他术语来描述这种表，表述方式不同而已，其本质都是相同的。——译者注



图2-1　交叉表实体关系图

我们来看看，使用交叉表是如何解决2.2节中描述的问题的。

2.5.1　通过账号查询产品和反过来查询

要查询指定账号的产品的所有属性，使用Products和Contacts表的联结查询是再简单不过的了：

Jaywalking/soln/join.sql

```
SELECT p.*
FROM  Products AS p JOIN Contacts AS c ON (p.account_id = c.a
WHERE c.account_id = 34;
```

有些人拒绝使用联结查询，他们认为那样很低效。然而，与2.2节中提出的方法相比，这个查询更好地使用了索引。

查询账号的细节也非常地简单，也方便优化。这里使用索引提高了联结查询的效率，而不是使用深奥的正则表达式：

Jaywalking/soln/join.sql

```
SELECT a.*
FROM   Accounts AS a JOIN Contacts AS c ON (a.account_id = c.
WHERE c.product_id = 123;
```

2.5.2　执行聚合查询

下面的例子返回每个产品相关的账号数量：

Jaywalking/soln/group.sql



```
SELECT product_id, COUNT(*) AS accounts_per_product
FROM Contacts
GROUP BY product_id;
```

获取每个账号相关的产品数量也很简单：

Jaywalking/soln/group.sql

```
SELECT account_id, COUNT(*) AS products_per_account
FROM Contacts
GROUP BY account_id;
```

生成其他更加复杂的报表也是可行的，比如找到相关账号最多的产品：

Jaywalking/soln/group.sql

```
SELECT c.product_id, c.accounts_per_product
FROM (
  SELECT product_id, COUNT(*) AS accounts_per_product
  FROM Contacts
  GROUP BY product_id
) AS c
HAVING c.accounts_per_product = MAX(c.accounts_per_product)
```

2.5.3 更新指定产品的相关联系人

你可以通过添加或者删除交叉表中的行来简单地修改字段中的条目。在Contacts表中，每条产品记录都是分开存储在不同的行中的，因而可以添加或删除。

Jaywalking/soln/remove.sql

```
INSERT INTO Contacts (product_id, account_id) VALUES (456, 34

DELETE FROM Contacts WHERE product_id = 456 AND account_id =
```

2.5.4 验证产品ID

你可以以另一张表中的合法数据为标准，使用外键来验证数据。通过声明Contacts. account_id引用Accounts.account_id，就能依靠数据库自身的约束来强制引用的完整性，以确保交叉表中只包含确凿存在的账号ID。

你还可以使用SQL的数据类型来约束条目。例如，设定字段中的条目应该是INTEGER或者DATE类型的，就可以确信所有条目都是合法的数据（不会是乱七八糟的像“banana”一样的数据）。

2.5.5　选择分隔符

你用不到分隔符了，因为数据都分开存储在不同的行中。即使条目中出现了逗号或者任何你可能想用做分隔符的其他字符，也不会有任何问题。

2.5.6　列表长度限制

至此，每个条目都位于交叉表中的独立的行内，列表的长度限制就变成了一张表可以实际存放的行数。如果可以限制条目总数，你应该在程序中加强对条目数量的使用，而不是统计列表的总体长度。

2.5.7　其他使用交叉表的好处

为Contacts.account_id做索引的查询效率比用逗号分隔列表中分串高效得多。在许多数据库中，声明某一列为外键会隐式地为该列创建索引（实际情况以相关文档为准）。

你还可以在交叉表中添加其他属性。比如，可以记录一个联系人被加入产品的具体日期，或产品的第一联系人及第二联系人。这些都是在逗号分隔的列表中无法做到的。

> 每个值都应该存储在各自的行与列中。

第3章　单纯的树

> 树就是树，你还需要考虑些什么呢？
>
> 罗纳德·里根

设想你正在为一个著名的科技新闻网站开发新版本。

这是一个很前卫的网站，因此，读者可以评论原文甚至相互回复，这样就某一主题的讨论又延伸出很多新的分支，其深度就会大大增加。你选择了一个简单的解决方案来跟踪这些回复分支：每条评论引用它所回复的评论。

Trees/intro/parent.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id)
);
```

很快，程序的逻辑就变得清晰起来，然而，要用一条简单的SQL语句检索一个很长的回复分支，还是很困难的。你只能获取一条评论的下一级或者联结第二级，到一个固定的深度。但这个贴子可以是无限深的，你可能需要执行很多次SQL查询才能获取给定主题的所有评论。

另一个你想到的主意是先获取一个主题的所有评论，然后再在程序的栈内存中将这些数据整合成你在学校里学到的传统的树形数据结构。但是，网站的产品人员告诉你，他们每天会发布数十篇文章，每篇文章可能有几百上千条评论，因此当每次有人访问网站都要做一次数据重整是不切实际的。

一定有一个更好的方法来存储评论的分支，同时可以简单而高效地获取一个完整的评论分支。

3.1 目标：分层存储与查询

存在递归关系的数据很常见，数据常会像树或者以层级方式组织。在树形结构中，实例被称为节点（node）。每个节点有多个子节点和一个父节点。最上层的节点叫根（root）节点，它没有父节点。最底层的没有子节点的节点叫叶（leaf）。而中间的节点简单地称为非叶（nonleaf）节点。

在层级数据中，你可能需要查询与整个集合或其子集相关的特定对象，例如下述树形数据结构。

组织架构图。职员与经理的关系是典型的树形结构数据，出现在无数的SQL书籍与论题中。在组织架构图中，每个职员有一个经理，在树结构中表现为职员的父节点。同时，经理也是一个职员。

话题型讨论。正如引言中介绍的，树形结构也能用来表示回复评论的评论链。在这种树中，评论的子节点是它的回复。

本章将用话题型讨论作为案例来讨论反模式及其解决方案。

3.2 反模式：总是依赖父节点

在很多书籍或文章中，最常见的简单解决方案是添加parent_id字段，引用同一张表中的其他回复。可以建一个外键约束来维护这种关系。下面是表的定义，图3-1是实例关系图。

Trees/anti/adjacency-list.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  parent_id    BIGINT UNSIGNED,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (parent_id) REFERENCES Comments(comment_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

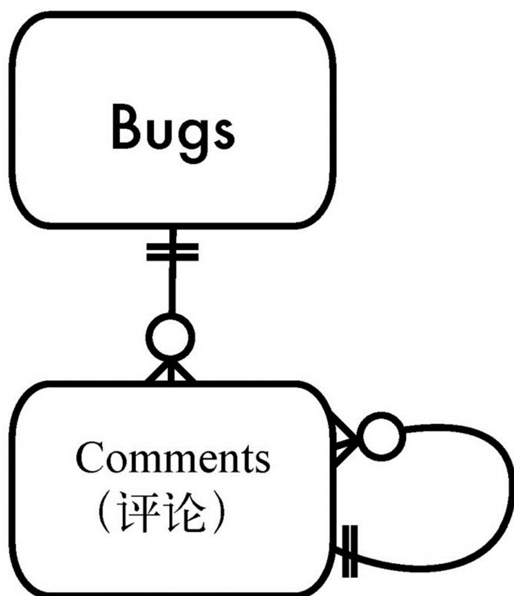


图3-1　邻接表的ERD

这样的设计叫做邻接表。这可能是程序员们用来存储分层结构数据中最普通的方案了。下表展示了一些简单的范例数据来显示评论表中的分层结构数据，同时图3-2展示了一棵该结构的树。

comment_id	parent_id	author	comment
1	NULL	Fran	这个Bug的成因是什么
2	1	Ollie	我觉得是一个空指针
3	2	Fran	不，我查过了
4	1	Kukla	我们需要查无效输入
5	4	Ollie	是的，那是个问题
6	4	Fran	好，查一下吧
7	6	Kukla	解决了

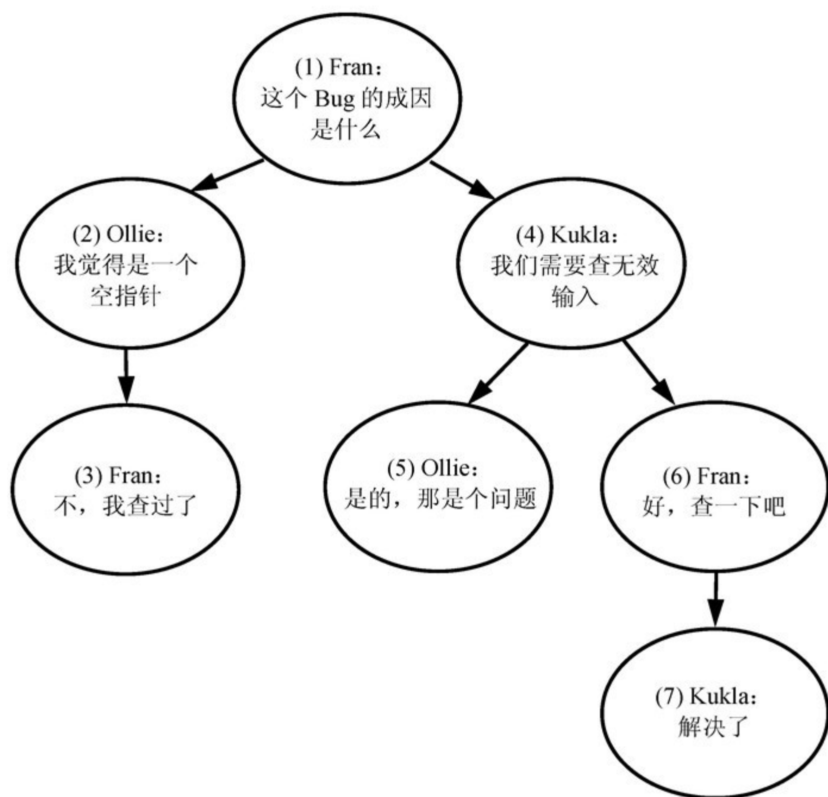


图**3-2**　线程化讨论示意图

3.2.1　使用邻接表查询树

但是，就算如此多的程序员会将邻接表作为默认的解决方案，它仍然可能成为一个反模式，原因在于它无法完成树操作中最普通的一项：查询一个节点的所有后代。你可以使用一个关联查询来获取一条评论和它的直接后代：

Trees/anti/parent.sql

```
SELECT c1.*, c2.*
FROM Comments c1 LEFT OUTER JOIN Comments c2
  ON c2.parent_id = c1.comment_id;
```

然而，这个查询只能获取两层的数据。树的特性就是它可以任意深地

扩展，因而你需要有方法来获取任意深度的数据。比如，可能需要计算一个评论分支的数量，或者计算一个机械设备所有部分的总开销。

当你使用邻接表的时候，这样的查询会变得很不优雅，因为每增加一层的查询都会需要额外扩展一个联结，而SQL查询中联结的次数是有上限的。如下的查询能够获得四层数据，但无法更多了：

Trees/anti/ancestors.sql

```
SELECT c1.*, c2.*, c3.*, c4.*
FROM Comments c1                          -- 1st level
  LEFT OUTER JOIN Comments c2
    ON c2.parent_id = c1.comment_id  -- 2nd level
LEFT OUTER JOIN Comments c3
  ON c3.parent_id = c2.comment_id  -- 3rd level
LEFT OUTER JOIN Comments c4
  ON c4.parent_id = c3.comment_id;  -- 4th level
```

这样的查询很笨拙，因为伴随着逐渐增加的后代层次，必须同等地增加联结查询的列。这使得执行一个聚合函数（比如COUNT()）变得极其困难。

另一种通过邻接表来获取树结构数据的方法是，先查询出所有行，在应用程序中自顶向下地重新构造出这棵树，然后再像树一样使用这些数据。

Trees/anti/all-comments.sql

```
SELECT * FROM Comments WHERE bug_id = 1234;
```

在处理数据之前就进行从数据库到应用程序之间的大量数据复制，是非常低效的，你可能仅仅需要一棵子树，而不是从根开始的完整的树；或者可能仅仅需要这些数据的聚合信息，比如评论的总数量（使用COUNT()函数）。

3.2.2 使用邻接表维护树

无可否认，一些操作通过邻接表来实现是非常方便的，比如增加一个叶子节点：

Trees/anti/insert.sql

```
INSERT INTO Comments (bug_id, parent_id, author, comment)
  VALUES (1234, 7, 'Kukla', 'Thanks!');
```

修改一个节点的位置或者一棵子树的位置也是很简单的：

Trees/anti/update.sql

```
UPDATE Comments SET parent_id = 3 WHERE comment_id = 6;
```

然而，从一棵树中删除一个节点会变得比较复杂。如果需要删除一棵子树，你不得不执行多次查询来找到所有的后代节点，然后逐个从最低级别开始删除这些节点以满足外键完整性。

Trees/anti/delete-subtree.sql

```
SELECT comment_id FROM Comments WHERE parent_id = 4; -- retur
SELECT comment_id FROM Comments WHERE parent_id = 5; -- retur
SELECT comment_id FROM Comments WHERE parent_id = 6; -- retur
SELECT comment_id FROM Comments WHERE parent_id = 7; -- retur

DELETE FROM Comments WHERE comment_id IN ( 7 );
DELETE FROM Comments WHERE comment_id IN ( 5, 6 );
DELETE FROM Comments WHERE comment_id = 4;
```

可以使用一个带ON DELETE CASCADE修饰符的外键约束来自动完成这些操作，只要能肯定确实是要删除这些节点，而不是修改它们的位置。

假如想要删除一个非叶子节点并且提升它的子节点，或者将它的子节点移动到另一个节点下，那么首先要修改子节点的parent_id，然后才能删除那个节点。

Trees/anti/delete-non-leaf.sql

```
SELECT parent_id FROM Comments WHERE comment_id = 6; -- retur

UPDATE Comments SET parent_id = 4 WHERE parent_id = 6;
```

```
DELETE FROM Comments WHERE comment_id = 6;
```

这些都是使用邻接表时需要多步操作才能完成的查询范例，你不得不写很多额外的代码，而其实数据库设计本身就能做得很简单和高效。

3.3　如何识别反模式

如果听到了如下的问题，可能你正在使用“单纯的树”这种反模式。

- “我们的树结构要支持多少层？”

  你正纠结于不使用递归查询获取一个给定节点的所有祖先或者后代。你做出让步，只支持有限层级数据的所有的操作，然后紧接着的一个很自然的问题就是：多少层才足够满足需求？

- “我总是很怕接触那些管理树结构的代码。”

  你已经采用了一种管理分层数据的比较复杂的方法，但使用的是错误的方法。每一项技术都会使得一些任务变得很简单，但通常是以其他的任务变得更复杂作为代价的。你可能选择了在应用程序设计中并不是最佳的方案来管理这些分层数据。

- “我需要一个脚本来定期地清理树中的孤立节点数据。”

  你的应用程序因为删除了非叶子节点而产生了一些迷失节点。在数据库中存储了一些复杂的结构时，需要确保在任何改变之后，这个结构都处在一致的、有效的状态下。你可以使用本章后面所描述的任何解决方案，同时配合触发器、级联外键等，来使得数据存储的结构更加适合项目需求，尽可能少地产生零碎的数据。

3.4　合理使用反模式

某些情况下，在应用程序中使用邻接表设计可能正好适用。邻接表设计的优势在于能快速地获取一个给定节点的直接父子节点，它也很容易插入新节点。如果这样的需求就是你的应用程序对于分层数据的全部操作，那使用邻接表就可以很好地工作了。

不要过度设计

> 我曾经为一个计算机数据中心写过一个库存跟踪程序。一些器材安装在电脑主机内部。比如缓存磁盘控制器是装在一个机架服务器里面的，扩展内存模块是装在磁盘控制器上的，等等。
>
> 我需要一个简单的数据库解决方案，来追踪那些包含了其他小部件的大设备的使用情况，同时也需要追踪每个独立部件的情况，并给出关于设备使用情况、折旧状态以及投资收益率的报表。
>
> 项目经理认为大的部件包含了一系列其他部件，同时这些部件还能再包含更小的部件，因而数据结构上这种层级关系可以是任意深度的。最终那花了我几个星期的时间来调整代码，以便让这棵树很好地适应数据库、用户界面、管理员界面和最终生成的报表。
>
> 然而在实际生产过程中，这个库存系统从来不曾有哪些设备间的关系达到两层嵌套，都是简单的父—子关系而已。如果我的最终用户能在一开始就确认两层的关系足够满足产品的需求，那么我们可以减少很大一部分的工作量。

某些品牌的数据库管理系统提供扩展的SQL语句，来支持在邻接表中存储分层数据结构。SQL-99标准定义了递归查询的表达式规范，使用`WITH`关键字加上公共表表达式。

> Trees/legit/cte.sql

```
WITH CommentTree
    (comment_id, bug_id, parent_id, author, comment, depth)
AS (
    SELECT *, 0 AS depth FROM Comments
    WHERE parent_id IS NULL
  UNION ALL
    SELECT c.*, ct.depth+1 AS depth FROM CommentTree ct
    JOIN Comments c ON (ct.comment_id = c.parent_id)
)
SELECT * FROM CommentTree WHERE bug_id = 1234;
```

Microsoft SQL Server 2005、Oracle 11_g_、IBM DB2和PostgreSQL 8.4都支持使用如上的查询表达式来进行递归查询。

和Oracle 10_g_一样，MySQL、SQLite和Infomix是少数几个暂时还不支持这种表达式的数据库，它们都被广泛地应用。也许我们可以假设

将来递归查询语法将被所有的主流数据库所支持，那么使用邻接表的设计也不会再受到这么多限制了。

Oracle 9_i_和10_g_也支持WITH子句，但并不是在递归查询时使用的。它们有着自己特殊的语法定义：START WITH和CONNECT BY PRIOR。

Trees/legit/connect-by.sql

```
SELECT * FROM Comments
START WITH comment_id = 9876
CONNECT BY PRIOR parent_id = comment_id;
```

3.5 解决方案：使用其他树模型

有几种方案可以代替邻接表模型，包括路径枚举、 嵌套集以及闭包表。接下来我将分三段来展示这三种设计方案是如何解决3.2节中所描述的存储和查询树型评论的问题的。

这些解决方案通常是这样的：首先可能看上去比邻接表复杂很多，但它们的确使得某些使用邻接表比较复杂或者很低效的操作变得更简单。如果你的应用程序确实需要提供这些操作，那么这些设计会是比邻接表更好的选择。

3.5.1 路径枚举

邻接表的缺点之一是从树中获取一个给定节点的所有祖先的开销很大。路径枚举的设计通过将所有祖先的信息联合成一个字符串，并保存为每个节点的一个属性，很巧妙地解决了这个问题。

路径枚举是一个由连续的直接层级关系组成的完整路径。如/usr/local/lib的UNIX路径是文件系统的一个路径枚举，其中usr是local的父亲，这也就意味着usr是lib的祖先。

在Comments表中，我们使用类型为VARCHAR的path字段来代替原来的parent_id字段。这个path字段所存储的内容为当前节点的最顶层的祖先到它自己的序列，就像UNIX的路径一样，你甚至可以使用‘/’作为路径中的分割符。

Trees/soln/path-enum/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  path         VARCHAR(1000),
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

comment_id	path	author	comment
1	1/	Fran	这个Bug的成因是什么
2	1/2/	Ollie	我觉得是一个空指针
3	1/2/3/	Fran	不，我查过了
4	1/4/	Kukla	我们需要查无效输入
5	1/4/5/	Ollie	是的，那是个问题
6	1/4/6/	Fran	好，查一下吧
7	1/4/6/7/	Kukla	解决了

你可以通过比较每个节点的路径来查询一个节点的祖先。比如，要找到评论#7——路径是1/4/6/7——的祖先，可以这样做：

Trees/soln/path-enum/ancestors.sql

```
SELECT *
FROM Comments AS c
WHERE '1/4/6/7/' LIKE c.path || '%';
```

这句查询语句会匹配到路径为1/4/6/%、1/4/%以及1/%的节点，而这些节点就是评论#7的祖先。

同时还可以通过将LIKE关键字两边的参数互换，来查询一个给定节点的所有后代。比如查找评论#4——路径为1/4——的所有后代，可以使用如下的语句：

Trees/soln/path-enum/descendants.sql

```
SELECT *
FROM Comments AS c
WHERE c.path LIKE '1/4/' || '%';
```

这句查询语句所找到的后代的路径分别是1/4/5、1/4/6以及1/4/6/7。

一旦你可以很简单地获取一棵子树或者从子孙节点到祖先节点的路径，就可以很简单地实现更多的查询，比如计算一棵子树所有节点上的值的总和（使用SUM聚合函数），或者仅仅是单纯地计算节点的数量。如果要计算从评论#4扩展出的所有评论中每个用户的评论数量，可以这样做：

Trees/soln/path-enum/count.sql

```
SELECT COUNT(*)
FROM Comments AS c
WHERE c.path LIKE '1/4/' || '%'
GROUP BY c.author;
```

插入一个节点也可以像使用邻接表一样地简单。可以插入一个叶子节点而不用修改任何其他的行。你所需要做的只是复制一份要插入节点的逻辑上的父亲节点的路径，并将这个新节点的ID追加到路径末尾就行了。如果这个ID是在插入时自动生成的，你可能需要先插入这条记录，然后获取这条记录的ID，并更新它的路径。比如，你使用的是MySQL，它的内置函数LAST_INSERT_ID()会返回当前会话的最新一条插入记录的ID，通过调用这个函数，便可以获得你所需要的ID，然后就可以通过新节点的父亲节点来获取完整的路径了。

Trees/soln/path-enum/insert.sql

```
INSERT INTO Comments (author, comment) VALUES ('Ollie', 'Good

UPDATE Comments
  SET path = (SELECT path FROM Comments WHERE comment_id = 7)
    || LAST_INSERT_ID() || '/'
WHERE comment_id = LAST_INSERT_ID();
```

路径枚举的方案也存在这一些缺点，比如就存在第2章“乱穿马路”中所描述的缺点：数据库不能确保路径的格式总是正确或者路径中的节点确实存在。依赖于应用程序的逻辑代码来维护路径的字符串，并且验证字符串的正确性的开销很大。无论将VARCHAR的长度设定为多大，依旧存在长度限制，因而并不能够支持树结构的无限扩展。

路径枚举的设计方式能够很方便地根据节点的层级排序，因为路径中分隔符两边的节点间的距离永远是1，因此通过比较字符串长度就能知道层级的深浅。

3.5.2 嵌套集

嵌套集解决方案是存储子孙节点的相关信息，而不是节点的直接祖先。我们使用两个数字来编码每个节点，从而表示这一信息，可以将这两个数字称为nsleft和nsright。

Trees/soln/nested-sets/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  nsleft       INTEGER NOT NULL,
  nsright      INTEGER NOT NULL,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs (bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

每个节点通过如下的方式确定nsleft和nsright的值：nsleft的数值小于该节点所有后代的ID，同时nsright的值大于该节点所有后代的ID。这些数字和comment_id的值并没有任何关联。

确定这三个值（nsleft，comment_id，nsrigh）的简单方法是对树进行一次深度优先遍历，在逐层深入的过程中依次递增地分配nsleft的值，并在返回时依次递增地分配nsright的值。

通过图3-3可以简单地想象出这样的分配方式。

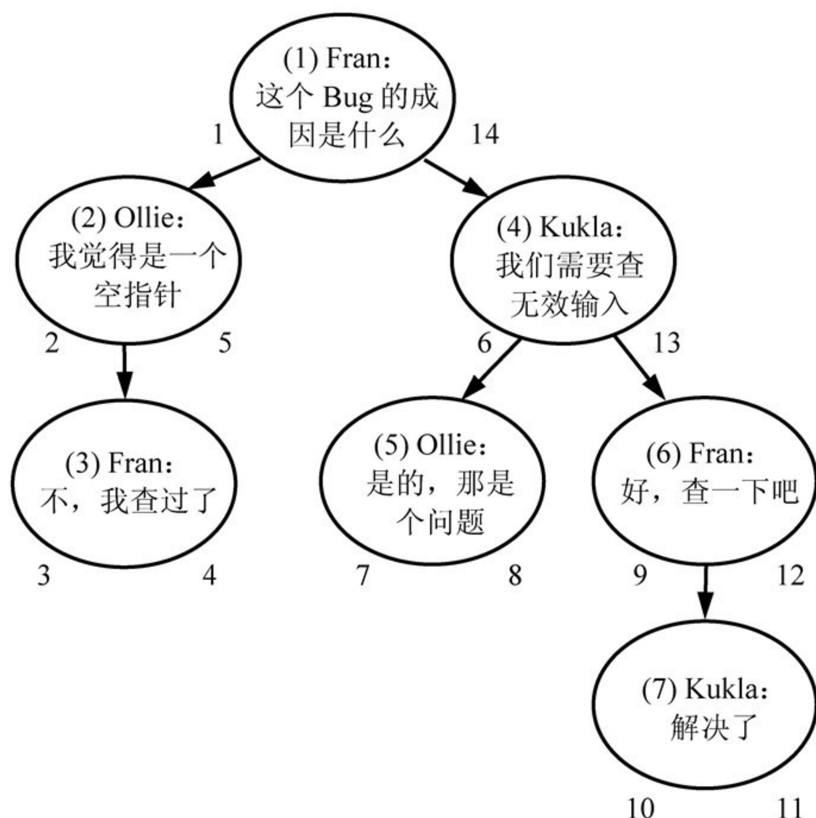


图3-3　嵌套集示意图

comment_id	nsleft	nsright	author	comment
1	1	14	Fran	这个Bug的成因是什么
2	2	5	Ollie	我觉得是一个空指针
3	3	4	Fran	不，我查过了
4	6	13	Kukla	我们需要查无效输入
5	7	8	Ollie	是的，那是个问题
6	9	12	Fran	好，查一下吧
7	10	11	Kukla	解决了

一旦你为每个节点分配了这些数字，就可以使用它们来找到给定节点的祖先和后代。比如，可以通过搜索哪些节点的ID在评论#4的`nsleft`和`nsright`范围之间来获取评论#4及其所有后代。

Trees/soln/nested-sets/descendants.sql

```
SELECT c2.*
FROM Comments AS c1
  JOIN Comments as c2
    ON c2.nsleft BETWEEN c1.nsleft AND c1.nsright
WHERE c1.comment_id = 4;
```

通过搜索评论#6的ID在哪些节点的`nsleft`和`nsright`范围之内，可以获取评论#6及其所有祖先：

Trees/soln/nested-sets/ancestors.sql

```
SELECT c2.*
FROM Comments AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 6;
```

使用嵌套集设计的主要优势便是，当你想要删除一个非叶子节点时，它的后代会自动地代替被删除的节点，成为其直接祖先节点的直接后代。尽管每个节点的左右两个值在示例图中是有序分配，而每个节点也总是和它相邻的父兄节点进行比较，但嵌套集设计并不必须保存分层关系。因而当删除一个节点造成数值不连续时，并不会对树的结构产生任何影响。

比如，你可以计算给定节点的深度然后删除它的父亲节点，随后，当你再次计算这个节点的深度的时候，它已经自动减少了一层。

Trees/soln/nested-sets/depth.sql

```
-- Reports depth = 3
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comment AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;

DELETE FROM Comment WHERE comment_id = 6;
```

```
-- Reports depth = 2
SELECT c1.comment_id, COUNT(c2.comment_id) AS depth
FROM Comment AS c1
  JOIN Comment AS c2
    ON c1.nsleft BETWEEN c2.nsleft AND c2.nsright
WHERE c1.comment_id = 7
GROUP BY c1.comment_id;
```

然而，某些在邻接表的设计中表现得很简单的查询，比如获取一个节点的直接父亲或者直接后代，在嵌套集的设计中会变得比较复杂。在嵌套集中，如果需要查询一个节点的直接父亲，我们会这么做：给定节点c1的直接父亲是这个节点的一个祖先，且这两个节点之间不应该有任何其他的节点，因此，你可以用一个递归的外联结来查询一个节点x，它即是c1的祖先，也同时是另一个Y节点的后代，随后我们使Y＝x并继续查询，直到查询返回空，即不存在这样的节点，此时的Y便是c1的直接父亲节点。

比如，要找到评论#6的直接父亲，你可以这样做：

Trees/soln/nested-sets/parent.sql

```
SELECT parent.*
FROM Comment AS c
  JOIN Comment AS parent
    ON c.nsleft BETWEEN parent.nsleft AND parent.nsright
  LEFT OUTER JOIN Comment AS in_between
    ON c.nsleft BETWEEN in_between.nsleft AND in_between.nsri
    AND in_between.nsleft BETWEEN parent.nsleft AND parent.ns
WHERE c.comment_id = 6
  AND in_between.comment_id IS NULL;
```

对树进行操作，比如插入和移动节点，使用嵌套集会比其他的设计复杂很多。当插入一个新节点时，你需要重新计算新插入节点的相邻兄弟节点、祖先节点和它祖先节点的兄弟，来确保它们的左右值都比这个新节点的左值大。同时，如果这个新节点是一个非叶子节点，你还要检查它的子孙节点。假设新插入的节点是一个叶子节点，如下的语句可以更新每个需要更新的地方：

Trees/soln/nested-sets/insert.sql

```
-- make space for NS values 8 and 9
UPDATE Comment
  SET nsleft = CASE WHEN nsleft >= 8 THEN nsleft+2 ELSE nslef
      nsright = nsright+2
WHERE nsright >= 7;

-- create new child of comment #5, occupying NS values 8 and
INSERT INTO Comment (nsleft, nsright, author, comment)
  VALUES (8, 9, 'Fran', 'Me too!');
```

如果简单快速地查询是整个程序中最重要的部分，嵌套集是最佳选择——比操作单独的节 点要方便快捷很多。然而，嵌套集的插入和移动节点是比较复杂的，因为需要重新分配左右值，如果你的应用程序需要频繁的插入、删除节点，那么嵌套集可能并不适合。

3.5.3 闭包表

闭包表是解决分级存储的一个简单而优雅的解决方案，它记录了树中所有节点间的关系，而不仅仅只有那些直接的父子关系。

在设计评论系统时，我们额外创建了一张叫做TreePaths的表，它包含两列，每一列都是一个指向Comments中comment_id的外键。

Trees/soln/closure-table/create-table.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);

CREATE TABLE TreePaths (
  ancestor    BIGINT UNSIGNED NOT NULL,
  descendant  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY(ancestor, descendant),
  FOREIGN KEY (ancestor) REFERENCES Comments(comment_id),
  FOREIGN KEY (descendant) REFERENCES Comments(comment_id)
);
```

我们不再使用Comments表来存储树的结构，而是将树中任何具有祖先—后代关系的节点对都存储在TreePaths表的一行中，即使这两个节点之间不是直接的父子关系；同时，我们还增加一行指向节点自己。更形象的表示可以参考图3-4。

祖先	后代	祖先	后代	祖先	后代
1	1	1	7	4	6
1	2	2	2	4	7
1	3	2	3	5	5
1	4	3	3	6	6
1	5	4	4	6	7
1	6	4	5	7	7

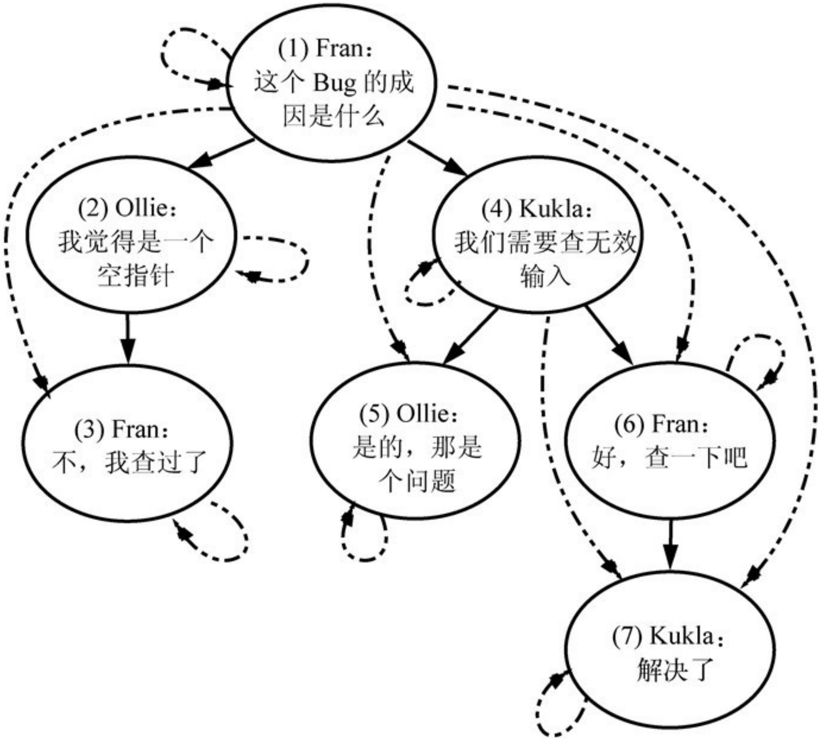


图3-4　闭包表示意图

通过TreePaths表来获取祖先和后代比使用嵌套集更加地直接。例如要获取评论#4的后代，只需要在TreePaths表中搜索祖先是评论#4的行就可以了：

Trees/soln/closure-table/descendants.sql

```
SELECT c.*
FROM Comments AS c
  JOIN TreePaths AS t ON c.comment_id = t.descendant
WHERE t.ancestor = 4;
```

要获取评论#6的所有祖先，只需要在TreePaths表中搜索后代为评论#6的行就可以了：

Trees/soln/closure-table/ancestors.sql

```
SELECT c.*
FROM Comments AS c
  JOIN TreePaths AS t ON c.comment_id = t.ancestor
WHERE t.descendant = 6;
```

要插入一个新的叶子节点，比如评论#5的一个子节点，应首先插入一条自己到自己的关系，然后搜索TreePaths表中后代是评论#5的节点，增加该节点和新插入节点的"祖先—后代"关系（包括评论#5的自我引用）：

Trees/soln/closure-table/insert.sql

```
INSERT INTO TreePaths (ancestor, descendant)
  SELECT t.ancestor, 8
  FROM TreePaths AS t
  WHERE t.descendant = 5
 UNION ALL
  SELECT 8, 8;
```

要删除一个叶子节点，比如评论#7，应删除所有TreePaths表中后代为评论#7的行：

Trees/soln/closure-table/delete-leaf.sql

```
DELETE FROM TreePaths WHERE descendant = 7;
```

要删除一棵完整的子树，比如评论#4和它所有的后代，可删除所有在

TreePaths表中后代为#4的行，以及那些以评论#4的后代为后代的行：

Trees/soln/closure-table/delete-subtree.sql

```
DELETE FROM TreePaths
WHERE descendant IN (SELECT descendant
                     FROM TreePaths
                     WHERE ancestor = 4);
```

请注意，如果你删除了TreePahts中的一条记录，并不是真正删除了这条评论。这对于评论系统这个例子来说可能很奇怪，但它在其他类型的树形结构的设计中会变得比较有意义。比如在产品目录的分类或者员工组织架构的图表中，当你改变了节点关系的时候，并不是真地想要删除一个节点。我们把关系路径存储在一个分开独立的表中，使得设计更加灵活。

要从一个地方移动一棵子树到另一地方，首先要断开这棵子树和它的祖先们的关系，所需要做的就是找到这棵子树的顶点，删除它的所有子节点和它的所有祖先节点间的关系。比如将评论#6从它现在的位置（评论#4的孩子）移动到评论#3下，首先做如下的删除（确保别把评论#6的自我引用删掉）。

Trees/soln/closure-table/move-subtree.sql

```
DELETE FROM TreePaths
WHERE descendant IN (SELECT descendant
                     FROM TreePaths
                     WHERE ancestor = 6)
  AND ancestor IN (SELECT ancestor
                   FROM TreePaths
                   WHERE descendant = 6
                     AND ancestor != descendant);
```

查询评论#6的祖先（不包含评论#6自身），以及评论#6的后代（包括评论#6自身），然后删除它们之间的关系，这将正确地移除所有从评论#6的祖先到评论#6和它的后代之间的路径。换言之，这就删除了路径(1, 6)、(1,7)、(4, 6)和(4, 7)，并且它不会删除(6, 6) 或 (6, 7)。

然后将这棵孤立的树和新节点及它的祖先建立关系。可以使用CROSS JOIN语句来创建一个新节点及其祖先和这棵孤立的树中所有节点间的笛卡儿积来建立所有需要的关系。

Trees/soln/closure-table/move-subtree.sql

```
INSERT INTO TreePaths (ancestor, descendant)
  SELECT supertree.ancestor, subtree.descendant
  FROM TreePaths AS supertree
    CROSS JOIN TreePaths AS subtree
  WHERE supertree.descendant = 3
    AND subtree.ancestor = 6;
```

这样就创建了评论#3及它的所有祖先节点到评论#6及其所有后代之间的路径。因此，新的路径是：(1, 6)、(2, 6)、(3, 6)、(1, 7)、(2, 7)、(3, 7)。同时，评论#6为顶点的这棵子树就成为了评论#3的后代。笛卡儿积能创建所有需要的路径，即使这棵子树的层级在移动过程中发生了改变。

闭包表的设计比嵌套集更加地直接，两者都能快捷地查询给定节点的祖先和后代，但是闭包表能更加简单地维护分层信息。这两个设计都比使用邻接表或者路径枚举更方便地查询给定节 点的直接后代和父代。

然而，你可以优化闭包表来使它更方便地查询直接父亲节点或子节点：在TreePaths表中增加一个path_length字段。一个节点的自我引用的path_length为0，到它直接子节点的path_length为1，再下一层为2，以此类推。查询评论#4的子节点就变得很直接：

Trees/soln/closure-table/child.sql

```
SELECT *
FROM TreePaths
WHERE ancestor = 4 AND path_length = 1;
```

3.5.4 你该使用哪种设计

每种设计都各有优劣，如何选择设计依赖于应用程序中的哪种操作最需要性能上的优化。在图3-5中，操作依据每种树的设计被标记为简单或者困难。你也可以参考以下列出的每种设计的优缺点。

设　计	表	查询子	查询树	插　入	删　除	引用完整性
邻接表	1	简单	困难	简单	简单	是
递归查询	1	简单	简单	简单	简单	是
枚举路径	1	简单	简单	简单	简单	否
嵌套集	1	困难	简单	困难	困难	否
闭包表	2	简单	简单	简单	简单	是

图3-5　层级数据设计比较

- 邻接表是最方便的设计，并且很多软件开发者都了解它。
- 如果你使用的数据库支持`WITH`或者`CONNECT BY PRIOR`的递归查询，那能使得邻接表的查询更为高效。
- 枚举路径能够很直观地展示出祖先到后代之间的路径，但同时由于它不能确保引用完整性，使得这个设计非常地脆弱。枚举路径也使得数据的存储变得比较冗余。
- 嵌套集是一个聪明的解决方案——但可能过于聪明了，它不能确保引用完整性。最好在一个查询性能要求很高而对其他需求要求一般的场合来使用它。
- 闭包表是最通用的设计，并且本章所描述的设计中只有它能允许一个节点属于多棵树。它要求一张额外的表来存储关系，使用空间换时间的方案减少操作过程中由冗余的计算所造成的消耗。

关于存储和操作SQL中的分层数据有很多东西可以说。Jeo Celko的*Trees and Hierarchies in SQL for Smarties*[Cel04]是一本介绍分层查询的好书，另一本讲解了树及图论的书是*SQL Design Patterns*[Tro06]，作者是Vadim Tropashko。后者更加正规及理论化。

> 一个分层数据结构包含了数据项和它们之间的关系。
>
> 需要合理的设计两者的模型来配合你的工作。

第4章　需要ID

> 外面的生物从猪看到人，从人看到猪，再从猪看到人，但它们已经分辨不出谁是猪谁是人了。
>
> 乔治·奥威尔，《动物庄园》

最近，有个程序员请教我一个很常见的问题：如何阻止表中出现重复项。 一开始，我认为可能是他的表中缺少一个主键，但后来发现并不是这么回事。

他的内容管理数据库中存储了在一个网站上所发表的文章。他使用一个交叉表来存储文章和标签这两张表之间的多对多的关系。

ID-Required/intro/articletags.sql

```
CREATE TABLE ArticleTags (
  id           SERIAL PRIMARY KEY,
  article_id   BIGINT UNSIGNED NOT NULL,
  tag_id       BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (article_id) REFERENCES Articles (id),
  FOREIGN KEY (tag_id)     REFERENCES Tags (id)
);
```

在他尝试通过给定的标签来查询文章数量的时候，返回了错误的结果。他知道“economy”标签只有5篇文章，但是查询结果告诉他有7篇。

ID-Required/intro/articletags.sql

```
SELECT tag_id, COUNT(*) AS articles_per_tag FROM ArticleTags
```

使用327这个tag_id去查询时，他看到这个标签和同一篇文章关联了三次，三条记录显示了同样的关系，尽管三条记录显示的关系id不同。

id	tag_id	article_id
22	327	1234
23	327	1234
24	327	1234

这张表是有主键的，但是主键并没有办法阻止像上表中那样的重复。有一种解决方案是对另外两列创建一个UNIQUE约束，但同时就会使得id这一列显得非常多余：为什么还需要它呢？

4.1 目标：建立主键规范

这章的目标就是要确认那些使用了主键，却混淆了主键的本质而造成的一种反模式。

每个了解数据库设计的人都知道，主键对于一张表来说是一个很重要，甚至必需的部分。这确实是事实，主键是好的数据库设计的一部分。主键是数据库确保数据行在整张表中唯一性的保障，它是定位到一条记录并且确保不会重复存储的逻辑机制。主键也同时可以被外键引用来建立表与表之间的关系。

难点是选择哪一列作为主键。大多数表中的每个属性的值都有可能被很多行使用。例如一个人的姓和名就一定会在表中重复出现，即使电子邮件地址或者美国社保编号或者税单编号也不能保证绝对不会重复。

在这样的表中，需要引入一个对于表的域模型无意义的新列来存储一个伪值。这一列被用作这张表的主键，从而通过它来确定表中的一条记录，即便其他的列允许出现适当的重复项。这种类型的主键列我们通常称其为伪主键或者代理键。

大多数的数据库提供一种和当前处理事务无关的底层方案，来确保每次都能生成全局唯一的一个整数作为伪主键，即使客户端此时正发起并发操作。

真的需要一个主键吗？

> 我曾经听一些开发人员声称他们的数据库表不需要主键。
>
> 有时候这些程序员想要避免想象中的维护唯一索引的开销，或者他们的表中并没有实现这一目的的列。
>
> 当你需要做下面这些事情的时候，主键约束是很重要的：
>
> - 确保一张表中的数据不会出现重复行；
> - 在查询中引用单独的一行记录；
> - 支持外键。
>
> 如果你不使用主键约束，就只有一个选择：检查是否有重复行。

```
SELECT bug_id FROM Bugs GROUP BY bug_id HAVING COUNT(*)
```

多久需要执行一次这样的检查？当你找到了一条重复记录时，又要如何处理？

一张没有主键的表就好像你的MP3播放列表里没有歌名一样。你依然可以听歌，但无法找到想听的那首歌，也没办法确保播放列表中没有重复的歌曲。

伪主键直到SQL:2003才成为一个标准，因而每个数据库都使用自己特有的SQL扩展来实现伪主键，甚至不同数据库中对于伪主键都有不同的名称（不同的表述），如下表。

特　　性	支持的数据库
AUTO_INCREMENT	MySQL
GENERATOR	Firebird, InterBase
IDENTITY	DB2, Derby, Microsoft SQL Server, Sybase
ROWID	SQLite
SEQUENCE	DB2, Firebird, Informix, Ingres, Oracle, PostgreSQL
SERIAL	MySQL, PostgreSQL

伪主键是非常有用的数据库特性，但并不是声明主键的唯一解决方案。

4.2　反模式：以不变应万变

很多的书、文章以及程序框架都会告诉你，每个数据库的表都需要一个主键，且具有如下三个特性：

- 主键的列名叫做id；
- 数据类型是32位或者64位整型；
- 主键的值是自动生成来确保唯一的。

在每张表中都存在一个叫做id的列是如此地平常，甚至id已经成为了主键的同义词。很多程序员在一开始学习SQL时就被灌输了错误的概念，认为主键就是像如下的程序那样定义的一列。

ID-Required/anti/id-ubiquitous.sql

```
CREATE TABLE Bugs (
  id          SERIAL PRIMARY KEY,
  description VARCHAR(1000),
  -- . . .
);
```

给每张表都增加一列id，使其使用显得太过随意。

4.2.1 冗余键值

你可能会发现在一张表中定义了id这一列作为主键，仅仅因为这么做符合传统，然而可能又同时存在另一列从逻辑上来说更为自然的主键，这一列甚至也具有UNIQUE约束。比如，在Bugs这张表中，程序会使用这个Bug所属项目的助记符或者其他的标识信息来标记一个Bug。

ID-Required/anti/id-redundant.sql

```
CREATE TABLE Bugs (
  id          SERIAL PRIMARY KEY,
  bug_id      VARCHAR(10) UNIQUE,
  description VARCHAR(1000),
  -- . . .
);

INSERT INTO Bugs (bug_id, description, ...)
  VALUES ('VIS-078', 'crashes on save', ...);
```

bug_id这一列和id有着相似的功能，都是为了唯一地标识一条记录。

4.2.2 允许重复项

一个组合键包含了多个不同的列。组合键的典型场景是在像BugsProducts这样的交叉表中。主键需要确保一个给定的bug_id和product_id的组合在整张表中只能出现一次，虽然同一个值可能在很多不同的配对中出现。

然而，当你使用id这一列作为主键，约束就不再是bug_id和product_id的组合必须唯一了。

ID-Required/anti/superfluous.sql

```
CREATE TABLE BugsProducts (
  id            SERIAL PRIMARY KEY,
  bug_id        BIGINT UNSIGNED NOT NULL,
  product_id    BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1), (1234, 1), (1234, 1); -- 重复项也是可以输入的
```

当你用这张交叉表去查询Bugs和Products的关系时，重复项会引起意料之外的结果。要确保没有重复项，你可以在id之外，额外声明另外两列需要一个UNIQUE约束。

ID-Required/anti/superfluous.sql

```
CREATE TABLE BugsProducts (
  id            SERIAL PRIMARY KEY,
  bug_id        BIGINT UNSIGNED NOT NULL,
  product_id    BIGINT UNSIGNED NOT NULL,
  UNIQUE KEY (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

但是，当你在bug_id和product_id这两列上应用了唯一性约束，id这一列就会变成多余的。

4.2.3 意义不明的关键字

单词“code”有很多意思，其中之一便是在交流中用于简化或者加密消息的。“code”还有一个意思就是写代码。在程序设计中，我们需要做的是尽量使得逻辑更加地清晰明了，而不是“使其复杂到难以辨认”。

“id”这个词是如此地普通，完全无法表达更深层次的意思，特别是在

你做两张表的联结查询，而它们都有一个叫做id的主键时。

ID-Required/anti/ambiguous.sql

```
SELECT b.id, a.id
FROM Bugs b
JOIN Accounts a ON (b.assigned_to = a.id)
WHERE b.status = 'OPEN';
```

如果你的程序用的是列名而不是列在表中的序号来编码，该如何区分缺陷id和账号id？在像PHP一样的动态语言中，这个问题显得更加突出。比如你需要获得一个关联数组的查询结果，除非在查询时指定了列别名，否则其中的一个id列会覆盖掉另一列id的值。

列名id并不会使查询变得更加清晰。但如果列名叫做bug_id或者account_id，事情就会变得更加简单。我们使用主键来唯一地定位一条记录，因此主键的列名就应该更加便于理解。

4.2.4 使用USING关键字

可能你很熟悉联结查询的语法，使用SQL关键字JOIN和ON来处理两张表的匹配数据行，就像下面的例子：

ID-Required/anti/join.sql

```
SELECT * FROM Bugs AS b JOIN BugsProducts AS bp ON (b.bug_id
```

SQL同时也支持另一种更加简洁的表达式来表示两张表的联结。如果两张表都有同样的列名，就可以用如下的表达式来重写上面的需求：

ID-Required/anti/join.sql

```
SELECT * FROM Bugs JOIN BugsProducts USING (bug_id);
```

然而，如果所有的表都要求定义一个叫做id的伪主键，那么作为外键的列将永远不能使用和引用的列相同的列名。同时，每次查询都必须使用啰嗦的ON表达式：

ID-Required/anti/join.sql

```
SELECT * FROM Bugs AS b JOIN BugsProducts AS bp ON (b.id = bp
```

4.2.5 使用组合键之难

一些开发人员拒绝使用组合键，因为他们觉得它太难以使用了。如果要比较两个键值，必须比较其包含的所有列的值；一个引用组合键的外键，其本身也必须是一个组合外键。此外，使用组合键需要打更多的字。

程序员拒绝在设计中使用组合键，就好像数学家拒绝使用二维或者三维坐标系，只使用一维数轴来计算所有现实事物一样。虽然那样的确会使得几何学变得简单，却无法描绘我们的真实世界。

特定范围序列

> 有些程序员通过给当前使用的最大值加1来获取一条新记录的ID：
>
> ```
> SELECT MAX(bug_id) + 1 AS next_bug_id FROM Bugs;
> ```
>
> 当客户端发起并发请求来获取新记录的id时，这样的做法并不可靠。同样的值可能被多个客户端同时获得，这样的情况称为“竞争”。
>
> 要避免由多客户端引起的竞争问题，就需要在计算并重新设定最大值时阻止并发的插入，你不得不锁住整张表——单纯锁住行并不够。锁住整张表的访问会造成系统性能的瓶颈，因为它使得所有的并发访问必须排队。
>
> 序列通过将运算和事务在逻辑上分离来解决并发问题。序列确保即使在多并发下，每次调用都会返回不同的值，因而无论是否需要将由序列返回的值插入新行，序列都不会再次生成同样的值。由于序列的这种特性，多个客户端可以同时发起请求，并且确信它们获取到的值在每个客户端中是唯一的。
>
> 大多数数据库都支持一些内置函数来获取一个序列生成的最后一个值。比如，MySQL中的这个函数叫做`LAST_INSERT_ID()`；Microsoft SQL Server使用叫做`SCOPE_IDENTITY()`的函数，Oracle中称为*`SequenceName`*`.CURRVAL()`。

> 这些函数都返回在自己的会话周期内生成的值，即使有其他的客户端同时在调用也是如此，因而不会产生竞争。

4.3　如何识别反模式

这章的反模式的特征很容易辨认：使用了过于普通的`id`作为表的主键的列名。实际上，绝无理由不使用另一个更加有意义的名称。

如果你遇到了下面的几个问题，可能是使用了这个反模式的征兆。

- “我觉得这张表不需要主键。”

  会这么说的开发人员一定是误解了“主键”和“伪主键”的含义。每张表都必须有一个主键来确保不出现重复项并定位每一行。他们可能想要一个更自然的列名来做主键或者需要一个组合键。

- “我怎么能在多对多的表中存储重复的项？”

  在一个多对多关系的交叉表中需要声明一个主键约束，或者至少需要有一个针对那些被引用为外键的列的唯一约束。

- “我学过数据库设计理论，里面说我应该把数据移到一张查询表中，然后通过ID查找。但是我不想这么做，因为每次我想要获得真实的数据，都不得不做一次联结查询。”

  这在数据库设计中是一个常见的误区，称为“正规化”（normalization），然而实际中对于伪主键并没有什么需要做的。更详细的信息参考附录A。

4.4　合理使用反模式

一些面向对象的框架假设“惯例优于配置”从而简化其设计。它们期望每张表都使用同样的方法来定义它的主键：使用`id`作为列名，并且使用类型为整型的伪主键。如果你使用了这样的一个框架，就可能不得不遵守这样的约定，才能够进一步使用这个框架所提供的其他特性。

使用伪主键，或者通过自动增长的整型的机制本身没有什么错误，但不是每张表都需要一个伪主键，更没有必要将每个伪主键都定义成`id`。

对于太长而不方便实现的自然键来说，伪主键是很好的代替品。比如在一个记录文件系统中所有文件属性的表中，文件路径是一个很好的自然键，但对一个字符串列做索引的开销会很大。

4.5　解决方案：裁剪设计

主键是约束而非数据类型。你可以定义任意列或任意多的列为主键，只要其数据类型支持索引。同时，还可以将一个列的数据类型定义为自增长的整型而不设定其为主键。这两者是完全无关的。

别被既有的惯例限制住设计。

4.5.1　直截了当地描述设计

为主键选择更有意义的名称：一个能够反应这个主键所代表的实体的类型的名字。比如， `Bugs`这张表的主键应该叫做`bug_id`。

外键应该尽可能地和所引用的列使用相同的名称，这通常意味着：一个主键的名称应该在整个数据库的设计中唯一；任意两张表都不应该使用相同的名称来定义主键，除非其中之一引用了另一个作为外键。然而，凡事都有例外，有时外键的名称需要和其所引用的主键区分开，从而使得它们之间的引用关系表现得更加清晰。

ID-Required/soln/foreignkey-name.sql

```
CREATE TABLE Bugs (
  -- . . .
  reported_by  BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
);
```

信息行业中存在一个叫做ISO/IEC 11179[1]的现行标准，用来描述元数据的命名惯例。换言之，这份标准就是用来指导你如何更合理地给数据库表中的每一列命名。就像大多数的ISO标准一样，这份标准文档也几乎是一份天书，但Joe Celko在他的*SQL Programming Style*[Cel05]一书中实践了这份标准。

[1] http://metadata-standards.org/11179/。

4.5.2 打破传统

面向对象的框架希望你使用id这个伪主键，但同时也允许无视这个规则转而使用别的名字。下面是Ruby on Rails的一个例子：

ID-Required/soln/custom-primarykey.rb

```
class Bug < ActiveRecord::Base
  set_primary_key "sbug_id"
end
```

一些开发人员认为仅仅在处理那些遗留数据库而无法使用自己所喜欢的规范时，才需要为主键列定义不同的名称，而事实上，即使对于新项目，为每一列指定一个有意义的名称也十分重要。

4.5.3 拥抱自然键和组合键

如果你的表中包含一列能确保唯一、非空以及能够用来定位一条记录，就别仅仅因为传统而觉得有必要再加上一个伪主键。

实践证明，一张表中的每一列都在最初的设计之后遭遇改变或者一开始就是不唯一的，这是再平常不过的事情。数据库的设计趋向于在整个项目的生命周期中不断地调整和优化，并且决策者也可能一点也不在乎自然键的“神圣不可侵犯”。有时侯，一个列在最开始时像是个很好的自然键，但随后又允许合法的重复项。此时，伪主键便成了唯一的选择。

在合适的时候也可以使用组合键，比如一条记录可以通过多列的组合完全定位，就像BugsProducts表，那就通过那些列创建一个组合键吧。

ID-Required/soln/compound.sql

```
CREATE TABLE BugsProducts (
  bug_id      BIGINT UNSIGNED NOT NULL,
  product_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (bug_id, product_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

```
INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1), (1234, 2), (1234, 3);
INSERT INTO BugsProducts (bug_id, product_id)
  VALUES (1234, 1); -- 错误：重复项
```

需要注意的是，一个引用了组合键的外键同样需要是一个组合键。这看上去像是很累赘地对其依赖的列做了一个副本，但这么做也有好处，它能简化原来需要一个联结查询才能获得的一条记录的相关属性。

> 规范仅仅在它有帮助时才是好的。

第5章　不用钥匙的入口

> 是故胜兵先胜而后求战，败兵先战而后求胜。
>
> 孙子

“比尔，我们实验室中的同一台服务器好像在同一天被两个经理预定了——这怎么可能？”测试实验室的经理冲进了我的小隔间说道：“你能查一下是怎么回事，然后解决一下这个问题吗？他们都对着我大吼大叫，说他们需要这台设备，并且说我耽误了他们的项目进度。”

几年前我使用MySQL设计了一个设备跟踪系统。MySQL默认的存储引擎是MyISAM，一个并不支持外键约束的东西。数据库的设计中有很多的逻辑关系，但无法保证引用完整性。

当程序不断地更新，并且使用了新的方法来操作数据的时候，我们制造了一个问题：当无法保障引用完整性时，报表中出现了不一致的情况，各部分的计算结果不一致，最终导致了同时预约的结果。

项目经理让我写一个监控脚本，然后让它在后台持续地执行，检查是否有不一致的情况发生，比如数据库中是否存在孤立的记录，并且当有此类错误发生时，通过电子邮件报警通知我们。

数据库中每个表与表的关系都需要使用这些脚本来进行检查。随着数据量和表的数目不断增长，监控脚本的查询量和相应的查询时间也随之增长，报警的邮件也变得很长。这样的场景，你是否似曾相识？

监控脚本工作得很好，但这显然是个很浪费的重复造轮子[1]工程。我所需要的是能找到一个方法，使得程序在用户提交了错误数据时能及时地发现。外键约束能做到吗？

> [1] reinvent the wheel：表示重复发明或开发已经存在的东西。——编者注

5.1　目标：简化数据库架构

关系数据库的设计基本上可以说就是关于每张独立表之间的关系的设计。引用完整性是合理的数据库设计和操作的非常重要的一部分。当一列或者多列声明了外键约束后，这些列中的数据必须在其父表（即所引用的表）的主键列或者唯一字段的列中存在。原理貌似很简单。

然而，一些开发人员不推荐使用引用完整性约束。你可能听说过这么几点不使用外键的原因。

- 数据更新有可能和约束冲突。
- 当前的数据库设计如此灵活，以致于不支持引用完整性约束。
- 数据库为外键建立的索引会影响性能。
- 当前使用的数据库不支持外键。
- 定义外键的语法并不简单，还需要查阅。

5.2　反模式：无视约束

即使第一感觉告诉你，省略外键约束能使得数据库设计更加简单、灵活，或者执行更加高效，你还是不得不在其他方面付出相应的代价——必须增加额外的代码来手动维护引用完整性。

5.2.1　假设无瑕代码

很多人对引用完整性的解决方案是通过编写特定的程序代码来确保数据间的关系的。每次插入新记录时，需要确保外键列所引用的值在其对应的表中存在；每次删除记录时，需要确保所有相关的表都要同时合理地更新。用时下流行的话来说就是：千万别犯错（make no mistakes）。

要避免在没有外键约束的情况下产生引用的不完整状态，需要在任何改变生效前执行额外的SELECT查询，以此来确保这些改变不会导致引用错误。比如，在插入一条新记录之前，需要检查对应的被引用记录是否存在：

Keyless-Entry/anti/insert.sql

```
SELECT account_id FROM Accounts WHERE account_id = 1;
```

然后才可以添加一个引用了这个账号的bug记录：

Keyless-Entry/anti/insert.sql

```
INSERT INTO Bugs (reported_by) VALUES (1);
```

要删除一条记录，需要先确认没有别的记录引用了该条记录：

Keyless-Entry/anti/delete.sql

```
SELECT bug_id FROM Bugs WHERE reported_by = 1;
```

随后才能删除这个账号：

Keyless-Entry/anti/delete.sql

```
DELETE FROM Accounts WHERE account_id = 1;
```

如果这个account_id为1的用户，恰巧在删除操作的查询和删除语句的执行间隙插入了一条新的缺陷记录，该怎么办？这看上去不太可能发生，但是戈登·莱顿（DOS 4的架构师）曾经说过一句很著名的话："不怕一万，就怕万一。"这样的确会造成一个破碎的引用关系——一个bug由一个不存在的账号提交。

唯一的做法是在检查数据时显式地锁住Bugs这张表，然后在删除账号完成之后再解锁。任何需要这样加锁的架构，在高并发和大数据量查询时的表现都非常糟糕。

5.2.2 检查错误

本章所描述的不正确的解决方案使用的是程序员写的外部脚本来检查错误的数据。

举例来说，在我们的缺陷数据库中，Bugs.status一列引用了BugStatus这张表。为了找到状态值有异常的缺陷记录，你可能会使用如下的查询语句：

Keyless-Entry/anti/find-orphans.sql

```
SELECT b.bug_id, b.status
FROM Bugs b LEFT OUTER JOIN BugStatus s
  ON (b.status = s.status)
WHERE s.status IS NULL;
```

可以想象一下，你需要为数据库中所有的引用关系写类似的脚本。

如果你发现自己正陷于使用这样的方法检查数据库中错误的引用关系的窘境时，下一个问题便是，多久需要执行一次这个脚本？每天手动执行成百上千或者更多次这样的查询脚本，是一件非常乏味的事情。

当你真的发现了一个错误的引用关系时，该怎么办？你能修复它吗？有时候也许行。比如，可以将一个无意义的值改成默认值[2]。

[2] SQL支持使用DEFAULT关键字。

Keyless-Entry/anti/set-default.sql

```
UPDATE Bugs SET status = DEFAULT WHERE status = 'BANANA';
```

不可避免，还有很多情况是无法通过简单设为默认值就能处理的。比如，Bugs.reported_ by这一列应该要引用一个提交这个缺陷的账号，而当这个值是无效值时，应该使用哪个账号来代替？

5.2.3 “那不是我的错！”

所有与数据库相关的代码都是完美的——这基本上是不可能的。你可以简单地使用几个函数来处理相似的数据更新，但是如果需要修改代

码的时候，要怎么保证应用程序中的每一个相关点都一起修改了呢？

而且可能有的用户直接操作了数据库，用了SQL查询工具或者使用了私有脚本。通过使用临时编写的SQL语句，很容易产生错误的引用。你应该假设这些事情会在你的应用程序生命周期的某一个时刻会发生。

你需要数据库中的数据保持连贯，这意味着，需要仰仗数据库中的引用关系始终是正常的，不出错的。但你不能确定所有的应用程序或者脚本在访问数据库时所作的事情都是正确合理的。

5.2.4 进退维谷

很多开发人员避免使用外键约束的理由，是因为这些约束会使得更新多张表中相关联的列变得比较麻烦。比如，如果你想要删除一条被其他记录所依赖的记录，就不得不删除所有的子记录来避免违反外键约束：

Keyless-Entry/anti/delete-child.sql

```
DELETE FROM BugStatus WHERE status = 'BOGUS'; -- ERROR!

DELETE FROM Bugs WHERE status = 'BOGUS';

DELETE FROM BugStatus WHERE status = 'BOGUS'; -- retry succee
```

你不得不为每张子表手动执行多条语句。如果你在将来的需求变更下又往数据库中添加了一张新表，就不得不修改所有相关的代码来删除这张新表中的数据。但这个问题是可以解决的。

而没解决的问题是，当你UPDATE一条被其他记录依赖的记录时，在没有更新父记录前，你不能更新子记录，而且也不能在更新父记录前更新子记录。你需要同步执行两边的更新，但是使用两个独立的更新语句是不现实的。这就是所谓的进退维谷。

Keyless-Entry/anti/update-catch22.sql

```
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS

UPDATE Bugs SET status = 'INVALID' WHERE status = 'BOGUS'; --
```

一些开发人员发现这样的情况几乎无法管理，因而他们决定干脆不使用外键。但是，我们稍后将会看到外键如何用一种简单而高效的方法同时更新和删除多张表中的记录。

5.3　如何识别反模式

如果你听到有人说的话像下面列举的这样，他们可能正使用了本章所描述的反模式。

- “我要怎么写这个查询语句来检查一个值是否没有同时在两张表中存在？”

  通常这样的需求是为了查找那些孤立的行。

- “有没有一种简单的方法来判断在一张表中存在的数据是否也在第二张表中存在？”

  这么做是用来确认父记录切实存在。外键会自动完成这些，并且外键会使用父表的索引尽可能高效地完成。

- “外键？有人告诉我别用它，因为那会影响数据库的效率。”

  性能总是用来裁剪设计的一个很好的理由，但总是会引入更多的问题，甚至包括性能问题本身。

5.4　合理使用反模式

有时你被迫使用不支持外键约束的数据库产品（比如MySQL的MyISAM存储引擎，或者比SQLite 3.6.19早的版本）。如果是这种情况，那你不得不使用别的方法来弥补，比如说前文描述的监控脚本。

同样也存在一些极度灵活的数据库设计，外键无法用来表示其对应的关系。如果你不能使用传统的引用完整性约束，很有可能你正在使用另一个SQL反模式。可以阅读第6章和第7章来获取更多的信息。

5.5　解决方案：声明约束

日语中有个短语poka-yoke，意思是“防差错技术”。这是一种制造工艺，通过在错误发生时对错误加以阻止、纠正或者引起注意来帮助消

除产品缺陷。这项工艺能够显著地提升产品质量，帮助减少纠错的必要，相比于使用这种工艺的开销，其所获得收益更高。

你可以将“防差错技术”的理论应用到你的数据库设计中——通过使用外键来确保引用完整性。相对于查找并修正完整性错误，你可以在进入数据库的第一道关卡上就阻止这样的错误发生。

Keyless-Entry/soln/foreign-keys.sql

```
CREATE TABLE Bugs (
  -- . . .
  reported_by         BIGINT UNSIGNED NOT NULL,
  status              VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

那些现存的代码以及临时的查询都将遵守同样的约束，因而，不会给任何被遗忘的代码片段或者其他的访问方式绕开约束的方法。数据库本身就会拒绝所有不合理的改变，无论这个改变是通过什么方式造成的。

通过使用外键，能够避免编写不必要的代码，同时还能确保一旦修改了数据库中的内容，所有的代码依旧能够用同样的方式执行。这节省了大量开发、调试以及维护时间。软件行业中每千行代码的平均缺陷数约为15~50个。在其他条件相同的情况下，越少的代码，意味着越少的缺陷。

5.5.1 支持同步修改

外键有另一个在应用程序中无法模拟的特性：级联更新。

Keyless-Entry/soln/cascade.sql

```
CREATE TABLE Bugs (
-- . . .
  reported_by         BIGINT UNSIGNED NOT NULL,
  status              VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
    ON UPDATE CASCADE
```

```
    ON DELETE RESTRICT,
  FOREIGN KEY (status) REFERENCES BugStatus(status)
    ON UPDATE CASCADE
    ON DELETE SET DEFAULT
);
```

这个解决方案允许你更新或者删除父记录，并且让数据库来处理那些引用了父记录的子记录。更新父表BugStatus和Accounts会自动地应用到Bugs表中的子记录。这就不会造成进退维谷的状况了。

在外键约束中声明ON UPDATE和ON DELETE的方式允许你控制级连操作的结果。比如说，在reported_by这个外键上声明的RESTRICT意味着你无法删除一个在Bugs这张表中被引用的账号。这个约束会阻止删除操作并且产生一个错误。而无论什么情况下，你删除一个status值，任何引用该值的记录都将被设置成默认值。

在执行更新和删除两个操作中的任意一个时，数据库都会自动修改两张表中的数据。外键的引用状态在操作之前和之后都将保持完好。

如果你往数据库中新加入一张子表，子表中的外键就规定了级联操作的行为。你不需要修改任何应用程序的代码。同时，无论多少张子表引用了同一张父表，都不需要改变任何东西。

5.5.2　系统开销过度？不见得

的确，外键约束需要多那么一点额外的系统开销，但相比于其他的一些选择，外键确实更高效一点。

- 不需要在更新或删除记录前执行SELECT进行检查。
- 在同步修改时不需要再锁住整张表。
- 不再需要执行定期的监控脚本来修正不可避免的孤立数据。

外键使用方便，提高性能，还能帮助你在任何简单或复杂形式的数据变更下始终维持引用完整性。

> 通过使用约束来帮助数据库防止错误。

第6章　实体—属性—值

> 如果你想把一只猫肢解了来研究它是怎么工作的，那么首先你要得到一只不工作的猫。
>
> 道格拉斯·亚当斯

“我要怎么按日期来统计记录条数？”对于数据库程序员来说，这是一个最基本的任务例子。它的解决方案在任何SQL入门介绍中都会出现，因为它包含了SQL最基本的语法：

> EAV/intro/count.sql

```
SELECT date_reported, COUNT(*)
FROM Bugs
GROUP BY date_reported;
```

然而，这个简单的解决方案需要两个假设。

- 所有的值都存在同一列中，比如`Bugs.data_reported`。
- 数据类型是可以比较的，因而`GROUP BY`可以通过比较两个值是否相等来分组。

假如这些假设不能满足呢？如果日期存储在`date_reported`或者`report_date`，或者其他任何列且每条记录的列名都不相同呢？如果日期的格式各式各样，而数据库无法简单地比较两个日期又该如何呢？

如果你在使用“实体—属性—值”反模式，就会遇到前面说的以及其他的一些问题。

6.1 目标：支持可变的属性

可扩展性是所有软件项目设计中最普遍的一个目标。我们都想设计出一个不需要过多修改，甚至不需要修改就能适合将来需求变更的软件。

这并不是一个新的课题，自1970年E. F. Codd在他的*A Relational Model of Data for Large Shared Data Banks*[Cod 70]一文中第一次介

绍了关系模型的概念以来，相似的关于关系数据模型不灵活性的争论就一直在持续。

通常来说，一张表由一些属性列组成，表中的每一条记录都使用这些列，因为每条记录表示的都是相似的对象实例。不同的属性集合表示不同的对象，因而就应该用不同的表来区分。

但是，在现代面向对象的编程模型中，不同的对象类型可能是相连的。比如，多个对象都可能是从同一个基类派生而来，它们既是实际子类的实例，也同时是父类的实例。我们可能想仅使用一张表来存储所有这些不同类型的对象，这样能方便进行比较和计算。但我们也需要将不同的子类分开存储，因为每个子类都有一些特殊的属性，和其他的子类甚至父类都不能共用。

我们继续使用Bugs数据库来举例。在图6-1中，`Bug`和`FeatureRequest`有一些公共属性，我们将其提炼为一个基类，称为`Issue`。每个事件都和一个报告它的人相关，同时也和一个产品相关，并且这个产品有个优先级用以比较。然而，`Bug`有一些独特的属性：产生错误的产品版本号和错误的级别。同样地，`FeatureRequest`也有自己的特有属性，比如，假设一个产品特性是和支持这一特性的开发赞助商相关的。

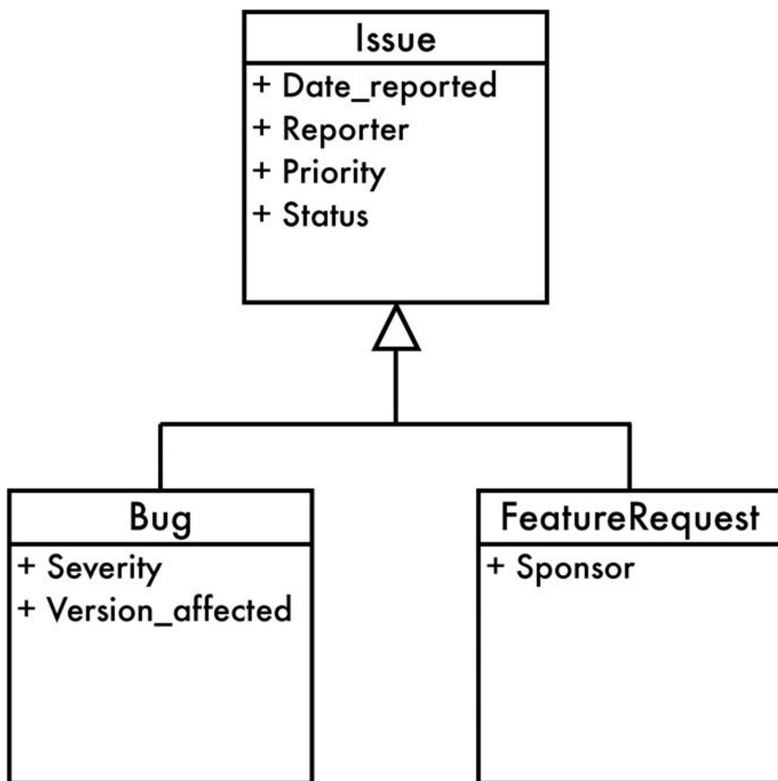


图6-1　**Bug**类型的类图

6.2　反模式：使用泛型属性表

对于某些程序员来说，当他们需要支持可变属性时，第一反应便是创建另一张表，将属性当成行来存储。图6-2中，属性表中的每条记录都包含三列。

- 实体：通常来说这就是一个指向父表的外键，父表的每条记录表示一个实体对象。
- 属性：在传统的表中，属性即每一列的名字，但在这个新的设计中，我们需要根据不同的记录来解析其标识的对象属性。
- 值：对于每个实体的每一个不同属性，都有一个对应的值。

比如，一个给定的`Bug`是一个实体对象，我们通过它的主键来标识

它，它的主键值为1234。这个对象有一个属性status，Bug1234的status的值为NEW。

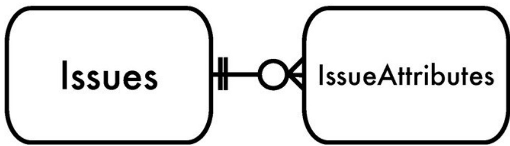


图6-2　EAV实体关系

这样的设计称为实体—属性—值，简称EAV。有时也称之为：开放架构、无模式或者名—值对。

EAV/anti/create-eav-table.sql

```
CREATE TABLE Issues (
  issue_id    SERIAL PRIMARY KEY
);

INSERT INTO Issues (issue_id) VALUES (1234);

CREATE TABLE IssueAttributes (
  issue_id    BIGINT UNSIGNED NOT NULL,
  attr_name   VARCHAR(100) NOT NULL,
  attr_value  VARCHAR(100),
  PRIMARY KEY (issue_id, attr_name),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);
INSERT INTO IssueAttributes (issue_id, attr_name, attr_value)
  VALUES
    (1234, 'product',          '1'),
    (1234, 'date_reported',    '2009-06-01'),
    (1234, 'status',           'NEW'),
    (1234, 'description',      'Saving does not work'),
    (1234, 'reported_by',      'Bill'),
    (1234, 'version_affected', '1.0'),
    (1234, 'severity',         'loss of functionality'),
    (1234, 'priority',         'high');
```

通过增加一张额外的表，可以获得如下这些好处。

- 这两张表的列都很少。

- 新增的属性不会对现有的表结构造成影响，不需要新增列。

- 避免了由于空值而造成的表内容混乱。

这看上去是一个改良过的设计，然而，设计上的简单化并不足以弥补其造成的使用上的难度。

6.2.1 查询属性

假设你的领导想要每天获取一份Bug的报表，在传统的表结构设计中，Issues表中仅包含了很简单的属性列，比如date_reported，要根据日期查询Bug记录，只需要执行一句很简单的语句：

EAV/anti/query-plain.sql

```
SELECT issue_id, date_reported FROM Issues;
```

要使用EAV设计来做相同的事情，就需要先从表IssueAttributes中提取出属性列为date_reported的所有记录。查询操作更加啰唆，而且不够清晰：

EAV/anti/query-eav.sql

```
SELECT issue_id, attr_value AS "date_reported"
FROM IssueAttributes
WHERE attr_name = 'date_reported';
```

6.2.2 支持数据完整性

使用了EAV的设计，需要放弃很多传统的数据库设计所带来的方便之处。

6.2.3 无法声明强制属性

要让你的领导能够顺利地生成项目报表，需要确保date_reported这个属性有值。在传统的数据库设计中，可以很简单地通过在声明的时候加上NOT NULL的限制来确保该列的值不为空。

在EAV的设计中，每个属性对应IssueAttributes表中的一行，而不是一列。你可能需要一个约束来检查对于每个issue_id都存在这么一行，并且这行的attr_name列的值是date_reported。

然而，SQL没有任何类型的约束支持这么做。因而你必须通过编写外部程序的代码确保这点。如果找到一个没有提交日期的Bug记录，应该为它加上一个日期吗？那应该给它赋什么值？如果胡乱猜测一个值或者使用默认值，对于你老板报表的精确性来说有多少影响？

6.2.4 无法使用SQL的数据类型

你的老板告诉你他的报表有点问题，因为人们输入的日期格式各式各样，甚至有时是个字符串而根本就不是一个日期。在传统的数据库中，你可以通过将一列的类型声明为DATE来确保这种情况不会发生。

EAV/anti/insert-plain.sql

```
INSERT INTO Issues (date_reported) VALUES ('banana'); -- ERRO
```

在EAV的设计中，IssueAttributes.attri_value列的数据类型就是一个单纯的字符串，从而才能仅用一列来适应任何可能的数据类型。因此，没有好办法来阻止无效数据的录入。

EAV/anti/insert-eav.sql

```
INSERT INTO IssueAttributes (issue_id, attr_name, attr_value)
  VALUES (1234, 'date_reported', 'banana'); -- Not an error!
```

有些人尝试扩展EAV的设计，为每一个SQL类型定义一个单独的attr_value列，不需要使用的列就留空。这可以让你使用SQL的数据类型，却使得查询变得更加恐怖：

EAV/anti/data-types.sql

```
SELECT issue_id, COALESCE(attr_value_date, attr_value_datetim
  attr_value_integer, attr_value_numeric, attr_value_float,
  attr_value_string, attr_value_text) AS "date_reported"
FROM IssueAttributes
```

```
WHERE attr_name = 'date_reported';
```

你可能需要添加更多的列来支持用户自定义的数据类型或者域（domain）。

6.2.5 无法确保引用完整性

在传统的数据库中，你可以定义一个指向另一张表的外键来约束某些属性的取值范围。比如，一个Bug或者事件的status属性应该是一张很小的BugStatus表中的一个值。

EAV/anti/foreign-key-plain.sql

```
CREATE TABLE Issues (
  issue_id          SERIAL PRIMARY KEY,
  -- other columns
  status            VARCHAR(20) NOT NULL DEFAULT 'NEW',
  FOREIGN KEY (status) REFERENCES BugStatus(status)
);
```

在EAV的设计中，你无法在attr_value列上使用这种约束方法。引用完整性的约束会应用到表中的每一行。

EAV/anti/foreign-key-eav.sql

```
CREATE TABLE IssueAttributes (
  issue_id          BIGINT UNSIGNED NOT NULL,
  attr_name         VARCHAR(100) NOT NULL,
  attr_value        VARCHAR(100),
  FOREIGN KEY (attr_value) REFERENCES BugStatus(status)
);
```

如果你像这样定义约束，那会强制表中每个属性的值都必须存在于BugStatus中，而不仅仅是status属性。

6.2.6 无法配置属性名

你老板的报表依旧非常不靠谱。你发现这些属性的命名不够清晰。有个Bug的属性叫做date_reported，但另一个Bug记录却把这个属性称作report_date。虽然两者都很清晰地表达了同样的意思。

既然如此，你又要如何计算每天的Bug数呢？

EAV/anti/count.sql

```
SELECT date_reported, COUNT(*) AS bugs_per_date
FROM (SELECT DISTINCT issue_id, attr_value AS date_reported
      FROM IssueAttributes
      WHERE attr_name IN ('date_reported', 'report_date'))
GROUP BY date_reported;
```

你又如何得知某条Bug记录没有用其他的名字来定义属性呢？你又如何得知某条Bug记录没有将同一个属性存储了两遍？你又要如何阻止这样的错误发生？

有一个解决方案是将attr_name列声明成一个外键，并指向一张存储着所有可能出现的属性名的表。然而，这不支持在运行时定义的各种属性，即使那是EAV设计的一个非常普遍的使用方式。

6.2.7　重组列

当数据是存储在一张传统的表中时，从Issues表中获取一整行记录，并得到一个议题的所有属性是个很平常的需求。

issue_id	date_reported	stats	优先级	描　述
1234	2009-06-01	NEW	HIGH	存储无法运行

由于每个属性在IssueAttributes表里都存储在独立的行中，要想像上面那样按行获取所有这些属性就需要执行一个联结查询，并将结果合并成行。同时，必须在写查询语句的时候就知道所有的属性名称。下面的查询语句重组了上表展示的行：

EAV/anti/reconstruct.sql

```
SELECT i.issue_id,
  i1.attr_value AS "date_reported" ,
  i2.attr_value AS "status",
  i3.attr_value AS "priority",
  i4.attr_value AS "description"
FROM Issues AS i
```

```
  LEFT OUTER JOIN IssueAttributes AS i1
    ON i.issue_id = i1.issue_id AND i1.attr_name = 'date_repo
  LEFT OUTER JOIN IssueAttributes AS i2
    ON i.issue_id = i2.issue_id AND i2.attr_name = 'status'
  LEFT OUTER JOIN IssueAttributes AS i3
    ON i.issue_id = i3.issue_id AND i3.attr_name = 'priority'
  LEFT OUTER JOIN IssueAttributes AS i4
    ON i.issue_id = i4.issue_id AND i4.attr_name = 'descripti
WHERE i.issue_id = 1234;
```

你必须使用外联结来进行查询，因为如果在所查询的这些属性中有任何一个不在IssueAttributes表中出现，则内联结会导致整个查询返回空记录。随着属性的数量不断增多，联结的数量也不断增长，查询的开销也成指数级地增长。

6.3 如何识别反模式

如果你听到项目团队发出了如下的疑问，很有可能就是使用了EAV反模式。

- “数据库不需要修改元数据就可以扩展。你还可以在运行时定义新的属性。”

  关系数据库不支持这种程度的灵活性。当某人声称能够设计一个可以任意扩展的数据库时，他基本上用的就是EAV的设计。

- “查询时我能用的最大数量的联结是多少？”

  如果你需要一个支持如此多联结的查询，并且联结的数量可能会达到数据库的限制时，你的数据库的设计可能是有问题的。而EAV的设计很有可能会导致这样的问题。

- “我想象不出怎么为我们的电子商务平台生成报告。我们需要雇一个顾问来做这事。”

  似乎很多现有的数据库驱动的软件使用EAV的设计来支持其强大的自定义能力，而这使得很多普通的报表查询变得极度复杂甚至不切实际。

6.4 合理使用反模式

在关系数据库中很难为EAV这个反模式正名，因为这就不得不放弃关系型范式的太多优点。但这不影响在某些程序中合理地使用这种设计来支持动态属性。

大多数应用程序仅仅在有限的几张表甚至于仅一张中需要存储无范式的数据，而其他的数据需求适用于标准的表设计。如果你明白在你的项目中使用EAV设计的风险和你要做的额外工作，并且谨慎地使用它，它的副作用会变得较小。但请一定要记住，那些富有经验的数据库顾问给出的报告显示，使用EAV设计的系统在一年以内就会变得极其笨重。

如果你有非关系数据管理的需求，最好的答案是使用非关系技术。这是一本关于SQL的书，而不是关于SQL选择的问题，因此我会简单地列出一些相关的技术。

- Berkeley DB是一个流行的Key-Value存储服务，非常容易被整合进入大部分程序。

  http://www.oracle.com/technology/products/berkeley-db/

- Cassandra是一个分布式的面向列的数据库，由Facebook开发，并提交给了Apache项目。

  http://incubator.apache.org/cassandra/

- CouchDB是一个面向文档的数据库——一个分布式的Key-Value存储系统，使用JSON编码数据。

  http://couchdb.apache.org/

- Hadoop和HBase组装了一个开源的DBMS，借助于Google的MapReduce算法为分布式大数据量查询提供分结构数据存储。

  http://hadoop.apache.org/

- MongoDB是一个像CouchDB一样的面向文档的数据库。

  http://www.mongodb.org/

- Redis是一个文件导向的内存数据库。

  http://code.google.com/p/redis

- Tokyo Cabinet是一个Key-Value存储结构，结合了POSIX DBM、GNU GDBM或Berkeley DB。

  http://1978th.net/

很多其他的非关系数据库项目也在不断地涌现。然而，在传统数据库中使用EAV设计的劣势也体现在这些非关系数据库上。当元数据不具有固定格式时，再简单的查询都会变得非常困难。上层应用就需要花费更多的时间、精力来组织数据结构。

6.5 解决方案：模型化子类型

如果EAV对于你的程序而言是正确的选择，你仍然需要在执行这个设计之前重新审视一遍。通过对列进行一些虽然老式但很好的分析，很可能会发现你的项目的数据可以更方便地被模型化到一个传统的表里，同时也提供了更保险的数据完整性支持。

除去使用EAV，还有好几个方法来存储这样的数据。当子类型数量有限时，大多数解决方案都能很好地工作，并且你知道每个子类型的属性。哪个解决方案最合适依赖于你查询数据的方式，因此你应该具体案例具体分析。

6.5.1 单表继承

最简单的设计是将所有相关的类型都存在一张表中，为所有类型的所有属性都保留一列。同时，使用一个属性来定义每一行表示的子类型。在这个例子中，这个属性称作issue_type。对于所有的子类型来说，既有一些公共属性，但同时又有一些子类型特有属性。这些子类型特有属性列必须支持空值，因为根据子类型的不同，有些属性并不需要填写，从而对于一条记录来说，那些非空的项会变得比较零散。

这个设计的名字来源于Martin Flower的一本著作：*Patterns of Enterprise Application Architecture*[Fow03]。

EAV/soln/create-sti-table.sql

```
CREATE TABLE Issues (
  issue_id          SERIAL PRIMARY KEY,
  reported_by       BIGINT UNSIGNED NOT NULL,
```

```
  product_id        BIGINT UNSIGNED,
  priority          VARCHAR(20),
  version_resolved  VARCHAR(20),
  status            VARCHAR(20),
  issue_type        VARCHAR(10),  -- BUG or FEATURE
  severity          VARCHAR(20),  -- only for bugs
  version_affected  VARCHAR(20),  -- only for bugs
  sponsor           VARCHAR(50),  -- only for feature requests
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

当程序需要加入新对象时，必须修改数据库来适应这些新对象。又由于这些新对象具有一些和老对象不同的属性，因而必须在原有表里增加新的属性列。可能会遇到一个很实际问题，就是每张表的列的数量是有限制的。

单表继承的另一个限制就是没有任何的元信息来记录哪个属性属于哪个子类型。在你的程序中，假如你知道有些属性并不适用于一个特定的行所表示的子类型对象，那么就可以忽略它们。但必须手动地跟踪哪些属性适用于哪些子类型。即使我们知道如果能用元数据在数据库中定义这些会更好，但也无能为力。

当数据的子类型很少，以及子类型特殊属性很少，并且你需要使用Active Record模式来访问单表数据库时，单表继承模式是最佳选择。

6.5.2　实体表继承

另一个解决方案是为每个子类型创建一张独立的表。每个表包含那些属于基类的共有属性，同时也包含子类型特殊化的属性。这个设计的名字来源于Martin Fowler的书。

EAV/soln/create-concrete-tables.sql

```
CREATE TABLE Bugs (
  issue_id          SERIAL PRIMARY KEY,
  reported_by       BIGINT UNSIGNED NOT NULL,
  product_id        BIGINT UNSIGNED,
  priority          VARCHAR(20),
  version_resolved  VARCHAR(20),
  status            VARCHAR(20),
```

```
  severity          VARCHAR(20), -- only for bugs
  version_affected VARCHAR(20), -- only for bugs
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE FeatureRequests (
  issue_id          SERIAL PRIMARY KEY,
  reported_by       BIGINT UNSIGNED NOT NULL,
  product_id        BIGINT UNSIGNED,
  priority          VARCHAR(20),
  version_resolved VARCHAR(20),
  status            VARCHAR(20),
  sponsor           VARCHAR(50),  -- only for feature requests
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

实体继承设计相比于单表继承设计的优势在于提供了一种方法，让你能阻止在一行内存储一些和当前子类型无关的属性。如果你引用一个并不存在于这张表中的属性列，数据库会自动提示你错误。比如，severity列并不在FeatureRequests表中：

EAV/soln/insert-concrete.sql

```
INSERT INTO FeatureRequests (issue_id, severity) VALUES ( ...
```

另一个使用实体继承表设计的好处便是，不用像在单表继承设计里那样使用额外的属性来标记子类型。

然而，很难将通用属性和子类特有的属性区分开来。因此，如果将一个新的属性增加到通用属性中，必须为每个子类表都加一遍。

没有元数据标记这些存储在自己表中的子类型相互之间有什么关系。那意味着，如果一个新来的程序员查看这些表定义，他只会注意到所有子类型的表中都有一些重复的列，但元信息没有告诉他任何有关这些表之间的关系，或者是否仅仅由于某种巧合，才使得这些表长得如此相似。

如果你希望不考虑子类型而在所有对象中进行过滤查找，问题会变得很复杂。如果想要将这件事情变得简单一点，就需要创建一个视图联

合这些表，仅选择公共的列。

EAV/soln/view-concrete.sql

```
CREATE VIEW Issues AS
  SELECT b.*, 'bug' AS issue_type
  FROM Bugs AS b
   UNION ALL
  SELECT f.*, 'feature' AS issue_type
  FROM FeatureRequests AS f;
```

当你很少需要一次性查询所有子类型时，实体继承表设计是最好的选择。

6.5.3 类表继承

第三个解决方案模拟了继承，把表当成面向对象里的类。创建一张基类表，包含所有子类型的公共属性。对于每个子类型，创建一个独立的表，通过外键和基类表相连。这个设计的名称同样来自于Martin Fowler的书。

EAV/soln/create-class-tables.sql

```
CREATE TABLE Issues (
  issue_id         SERIAL PRIMARY KEY,
  reported_by      BIGINT UNSIGNED NOT NULL,
  product_id       BIGINT UNSIGNED,
  priority         VARCHAR(20),
  version_resolved VARCHAR(20),
  status           VARCHAR(20),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE Bugs (
  issue_id         BIGINT UNSIGNED PRIMARY KEY,
  severity         VARCHAR(20),
  version_affected VARCHAR(20),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);
```

```
CREATE TABLE FeatureRequests (
  issue_id          BIGINT UNSIGNED PRIMARY KEY,
  sponsor           VARCHAR(50),
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id)
);
```

基类表和子类表之间一对一的关系由元数据来确保，因为子类型表的外键也同样是主键，因而就必须是唯一的。这个解决方案提供了一个高效的方法来查询所有的记录，因为你仅仅查询基类的属性。一旦你找到了合适的记录，就可以通过查询对应的子类型表来获取子类型特殊化的属性。

你不需要了解在基类表中的行表示的是哪个子类型，如果仅有很少的子类型，那么可以写一个联结查询来一次性获取所有的记录，产生一个像单表继承设计里的那样稀疏结果集。当这个子类型不具有某个属性时，其值是空的。

EAV/soln/select-class.sql

```
SELECT i.*, b.*, f.*
FROM Issues AS i
  LEFT OUTER JOIN Bugs AS b USING (issue_id)
  LEFT OUTER JOIN FeatureRequests AS f USING (issue_id);
```

这也是定义一个视图的好方法。

当你经常要查询所有子类型时这个设计是最佳选择，引用这些公共列就行了。

6.5.4 半结构化数据模型

如果你有很多子类型或者你必须经常地增加新的属性支持，那么可以使用一个BLOB列来存储数据，用XML或者JSON格式——同时包含了属性的名字和值。Martin Fowler称这个模式为：序列化大对象块（Serialized LOB）。

EAV/soln/create-blob-tables.sql

```
CREATE TABLE Issues (
  issue_id          SERIAL PRIMARY KEY,
```

```
  reported_by       BIGINT UNSIGNED NOT NULL,
  product_id        BIGINT UNSIGNED,
  priority          VARCHAR(20),
  version_resolved VARCHAR(20),
  status            VARCHAR(20),
  issue_type        VARCHAR(10),   -- BUG or FEATURE
  attributes        TEXT NOT NULL, -- all dynamic attributes f
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

这个设计的优势之处就在于其优异的扩展性。你可以在任何时候，将新的属性添加到blob字段中。每行存储一个完整的属性集合，因此你可以有尽可能多的子类型，有多少行就可以有多少个。

相应地，该设计的缺点就是在这样的一个结构中，SQL基本上没有办法获取某个指定的属 性。你不能在一行blob字段中简单地选择一个独立的属性，并对其进行限制、聚合运算、排序等其他操作。你必须获取整个blob字段结构并通过程序去解码并且解释这些属性。

当你不能将需求和设计限制在一个有限的子类型集合中，或者当你需要绝对的灵活性以在任何时间调整属性时，这个方案就是最佳选择。

6.5.5　后处理

遗憾的是，有时你不得不使用EAV设计，比如你接手了一个项目，但不能改变它的原始设计，或者你的公司获得了一个第三方的软件，并且恰巧使用的是EAV。如果是这样的情况，请牢记在6.2节中我们所描述的那些问题，从而你可以有所准备并且计划好额外的工作来让这个设计工作良好。

综上所述，别尝试像在传统表中那样写查询语句，将实体当成单行数据读取。取而代之的是，查询和实体关联的属性并且将这些行组合在一起，就像它们存储的结构一样。

EAV/soln/post-process.sql

```
SELECT issue_id, attr_name, attr_value
FROM IssueAttributes
WHERE issue_id = 1234;
```

查询的结果应该如下表：

issue_id	attr_name	attr_value
1234	date_reported	2009-06-01
1234	description	Saving does not work
1234	priority	HIGH
1234	product	Open RoundFile
1234	reported_by	Bill
1234	severity	loss of functionality
1234	status	NEW

这个查询对你来说写起来很容易，对数据库来说执行起来也很容易。即使当你写这个查询的时候并不知道有多少相关属性，它依旧会返回和这个事件相关的所有属性。

要使用这种格式的结果，你需要在程序中写一段代码来遍历结果集中的每一行记录，并且设置程序中对象的属性。下面有个PHP的代码范例：

EAV/soln/post-process.php

```
<?php

$objects = array();

$stmt = $pdo->query(
    "SELECT issue_id, attr_name, attr_value
     FROM IssueAttributes
     WHERE issue_id = 1234");

while ($row = $stmt->fetch()) {
  $id    = $row['issue_id'];
  $field = $row['attr_name'];
  $value = $row['attr_value'];
  if (!array_key_exists($id, $objects)) {
    $objects[$id] = new stdClass();
  }

  $objects[$id]->$field = $value;
}
```

这看上去有太多的工作要做，但这就是使用像EAV这样的一个系统套

系统结构所造成的必然结果。

SQL已经提供了一个方法来明确地定义属性——在明确的列中。使用EAV设计，你让SQL使用新的方法来定义属性，因此SQL对于这种方法的支持是如此笨拙和低效也不足为奇了。

> 为元数据使用元数据。

第7章　多态关联

> 的确，有些人是两面派。
>
> 稻草人，《绿野仙踪》

让我们允许用户对Bug记录进行评论。一个给定的Bug可能会有很多评论，但任何的评论都只针对一个Bug记录。因此，在Bug和评论之间是一对多的关系。这种简单的关系如图7-1所示，接下来的SQL脚本显示如何创建这张表。

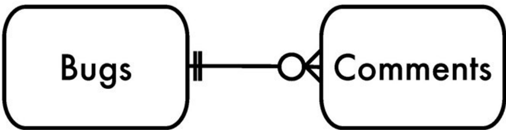


图7-1　简单关系

> Polymorphic/intro/comments.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  bug_id       BIGINT UNSIGNED NOT NULL,
  author_id    BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME NOT NULL,
  comment      TEXT NOT NULL,
  FOREIGN KEY (author_id) REFERENCES Accounts(account_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

但是，可以进行评论的表可能会有两张，Bugs和FeatureRequests是类似的实体，尽管你可能分开存储它们（参见6.5节）。你想要在单表中存储所有的评论，不关心它们的类型——无论是Bug还是新特性——但你不能声明一个指向多张表的外键。如下的定义是无效的：

Polymorphic/intro/nonsense.sql

```
...
FOREIGN KEY (issue_id)
    REFERENCES Bugs(issue_id) OR FeatureRequests(issue_id)
);
```

有些开发人员还尝试着写如下的SQL语句查询多张表，却是无效的：

Polymorphic/intro/nonsense.sql

```
SELECT c.*, i.summary, i.status
FROM Comments AS c
JOIN c.issue_type AS i USING (issue_id);
```

SQL不支持按行联结不同的表。SQL的语法要求提交查询时就明确写明所有的表名。查询过程中表名不能修改。这样的情况问题出在哪里，要怎么解决呢？

7.1 目标：引用多个父表

在“绿野仙踪”里，当多萝西询问稻草人她应该走哪条路才能到翡翠城时，稻草人指给了她不确定的方向。本应该是非常简单的问题，但当稻草人一次指了两条路给她时，多萝西迷惑起来。

图7-2描绘了在实体关系中的这种令人迷惑的联系。子表中的外键“分叉”了，因此Comments表中的一条记录即可能匹配Bugs表中的某条记录，也可能匹配于FeatureReqeuests表中的某条记录。

图中的弧线表明这是一个排他选择：一个给定的评论只能引用一个Bug或者一个特性。

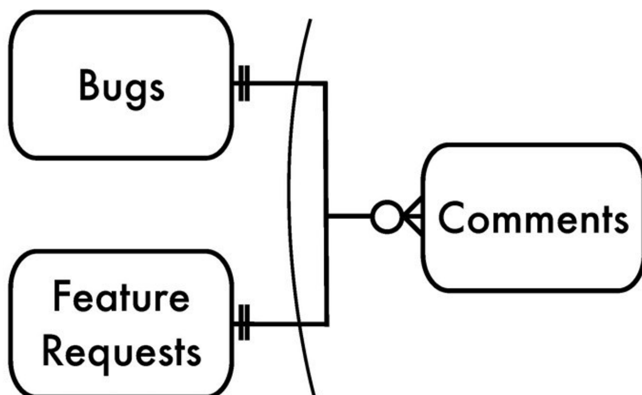


图7-2　多态关联

7.2　反模式：使用双用途外键

有一个解决方案已经流行到足以正式命名了，那就是：多态关联。有时候也叫做杂乱关联，因为它可以同时引用多个表。

7.2.1　定义多态关联

为了使多态关联能够正常工作，在issue_id这个外键之外你必须再添加一列，这个额外的列记录了当前行所引用的表名。在这个例子中，这个额外的列称为issue_type，取值范围是Bugs或者FeatureRequests。

Polymorphic/anti/comments.sql

```
CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  issue_type   VARCHAR(20),      -- "Bugs" or "FeatureRequests"
  issue_id     BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME,
  comment      TEXT,
  FOREIGN KEY (author) REFERENCES Accounts(account_id)
);
```

你立刻就可以发现一个区别：issue_id上的外键定义不见了。事实

上，由于一个外键必须指定一个确切的表，使用多态关联就意味着无法在元数据中定义这样的关系。因而，就没有任何保障数据完整性的手段来确保Comments.issue_id中的值在其父表中存在。

同样地，没有元数据来确保Comments.issue_type中的值确实对应于数据库中存在的一张表。

> 混合数据与元数据
>
> 你可能已经注意到了多态关联和前一章的EAV反模式有着相似的特征。在这两个反模式中，元数据对象的名字是存储在字符串中的。在EAV中，属性列的名字是以字符串的格式存储在attr_name列中。在多态关联中，父表的名字是存储在issue_type列中的。有时这样的设计被称为：混合数据与元数据。第8章会有更详细的概念解释。

7.2.2　使用多态关联进行查询

在Comments表中的一个issue_id可能同时出现在Bugs和FeatureRequests这两张父表中，或者只出现在其中一张表里。因此，在使用issue_type进行联结查询时，就必须格外谨慎，必须确保不会出现明明是要查找Bugs的评论，却获得了FeatureRequests的评论这种情况。

比如，如下的查询目的在于获取id为1234的这个Bug的所有评论：

> Polymorphic/anti/select.sql

```
SELECT *
FROM Bugs AS b JOIN Comments AS c
  ON (b.issue_id = c.issue_id AND c.issue_type = 'Bugs')
WHERE b.issue_id = 1234;
```

当所有的Bug记录都是存在单个的Bugs表中时，上面的这个查询能够正常地工作，但当Comments同时和Bugs表以及FeatureRequests表相关联时，就会出现问题。SQL的语法规定，在联结查询时必须指明所有需要查询的表，没办法在查询过程中根据Comments.issue_type的值来切换不同的表。

要查找一条给定的评论对应的Bug记录或者特性需求，需要执行一条

同时外联Bugs和FeatureRequests两张表的查询。仅有一张表满足这个查询需求，因为联结查询的条件依赖于Comment.issue_type中的值。使用外联结意味着结果中那些来自于非匹配的字段将为空值。

Polymorphic/anti/select.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN Bugs AS b
    ON (b.issue_id = c.issue_id AND c.issue_type = 'Bugs')
  LEFT OUTER JOIN FeatureRequests AS f
    ON (f.issue_id = c.issue_id AND c.issue_type = 'FeatureRe
```

结果看起来类似下面这样。

c.comment_id	c.issue_type	c.issue_id	c.comment	b.issue_id	f.issue_id
6789	Bugs	1234	It crashes!	1234	NULL
9876	Feature...	2345	Great idea!	NULL	2345

7.2.3 非面向对象范例

在Bugs和FeatureRequests的例子中，这两张表代表了模型相关的子类型。多态关联也同样可以用在那些父表间完全无关的设计中。比如，在一个电子商务数据库中，Users（用户）和Orders（订单）这两张表可能都和Addresses（地址）相关，如图7-3所示。

Polymorphic/anti/addresses.sql

```
CREATE TABLE Addresses (
  address_id    SERIAL PRIMARY KEY,
  parent        VARCHAR(20),      -- "Users" or "Orders"
  parent_id     BIGINT UNSIGNED NOT NULL,
  address       TEXT
);
```

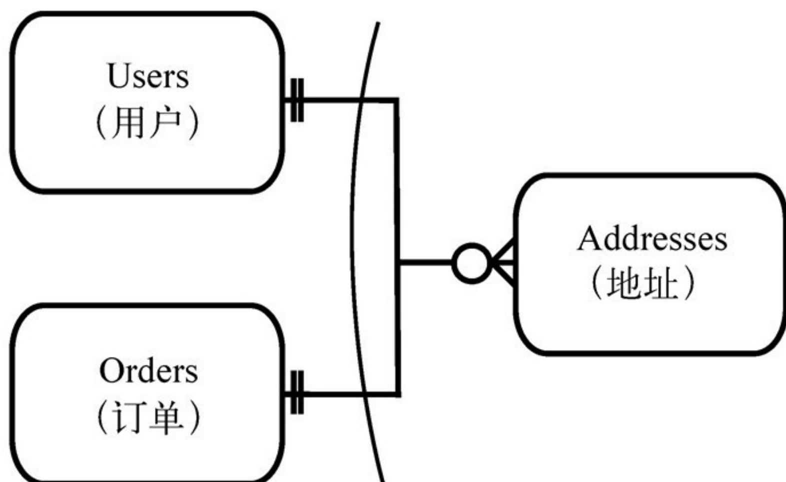


图7-3　地址多态关联

在这个例子中，Addresses表有一个多态列，对于给定的一条地址记录，其父表的值为Users或者Orders，两者二选其一。一个给定的地址不能既和用户相关，又和订单相关，即使订单可能是由一个用户创建然后邮寄给他自己的。

此外，如果一个用户的收货地址也是账单地址，你应该在Addresses表中将其区分开来;同样地，任何其他的父表都需要留意Addresses表中不同的地址字段的特殊用处。这些说明信息将像野草一样疯长。

Polymorphic/anti/addresses.sql

```
CREATE TABLE Addresses (
  address_id    SERIAL PRIMARY KEY,
  parent        VARCHAR(20),     -- "Users" or "Orders"
  parent_id     BIGINT UNSIGNED NOT NULL,
  users_usage   VARCHAR(20),     -- "billing" or "shipping"
  orders_usage  VARCHAR(20),     -- "billing" or "shipping"
  address       TEXT
);
```

7.3　如何识别反模式

如果你听到类似下面的这些说法，那有可能就使用了多态关联的反模式了。

- “这种标记架构可以让你将标记（或者其他属性）和数据库中的任何其他资源联系起来。”

  就像EAV的设计一样，你应该怀疑任何声称有无限扩展性的设计。

- “你不能在我们的数据库设计中声明外键。”

  这是另一个危险信号。外键是关系数据库的基本特征，一个没有适当的引用完整性的设计会有很多的问题。

- “entity_type这列是干嘛的？哦，那个列是用来告诉你这条记录的其他列是和什么东西相关的。”

  任何外键都强制一张表中所有的行引用同一张表。

Ruby on Rails的框架通过声明带有:polymorphic属性的Active Record类来支持多态关联。比如，你可以使用如下方式将Comments和Bugs以及FeatureRequests关联起来。

> Polymorphic/recog/commentable.rb

```
class Comment < ActiveRecord::Base
  belongs_to :commentable, :polymorphic => true
end

class Bug < ActiveRecord::Base
  has_many :comments, :as => :commentable
end

class FeatureRequest < ActiveRecord::Base
  has_many :comments, :as => :commentable
end
```

Java Hibernate框架使用大量的模式定义来支持多态关联。

7.4　合理使用反模式

你应该尽可能地避免使用多态关联——应该使用外键约束等来确保引用完整性。多态关联通常过度依赖上层程序代码而不是数据库的元数据。

当你使用一个面向对象的框架（诸如Hibernate）时，多态关联似乎是不可避免的。这种类型的框架通过良好的逻辑封装来减少使用多态关联的风险。如果你选择了一个成熟、有信誉的框架，那可以相信框架的作者已经完整地实现了相关的逻辑代码，不会造成错误。但如果你不使用框架而自己从头开始实现，那就真有重新发明轮子之嫌了。

7.5　解决方案：让关系变得简单

既要避免多态关联的缺点，又同时支持你所需要的数据模型，最好的选择是重新设计数据库。接下来几节介绍的解决方案能够完全满足我们所需的数据关系，同时又使用数据库的元数据来确保数据及引用的完整性。

7.5.1　反向引用

当你看清楚问题的根源时，解决方案将变得异常的简单：多态关联是一个反向关联。

7.5.2　创建交叉表

Comments表中的外键无法同时引用多张父表，因而，我们使用多个外键同时引用Comments表即可。为每个父表创建一张独立的交叉表，每张交叉表同时包含一个指向Comments的外键和一个指向对应父表的外键，如图7-4所示。

Polymorphic/soln/reverse-reference.sql

```
CREATE TABLE BugsComments (
  issue_id     BIGINT UNSIGNED NOT NULL,
  comment_id   BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES Bugs(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);

CREATE TABLE FeaturesComments (
```

```
  issue_id    BIGINT UNSIGNED NOT NULL,
  comment_id  BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES FeatureRequests(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
```

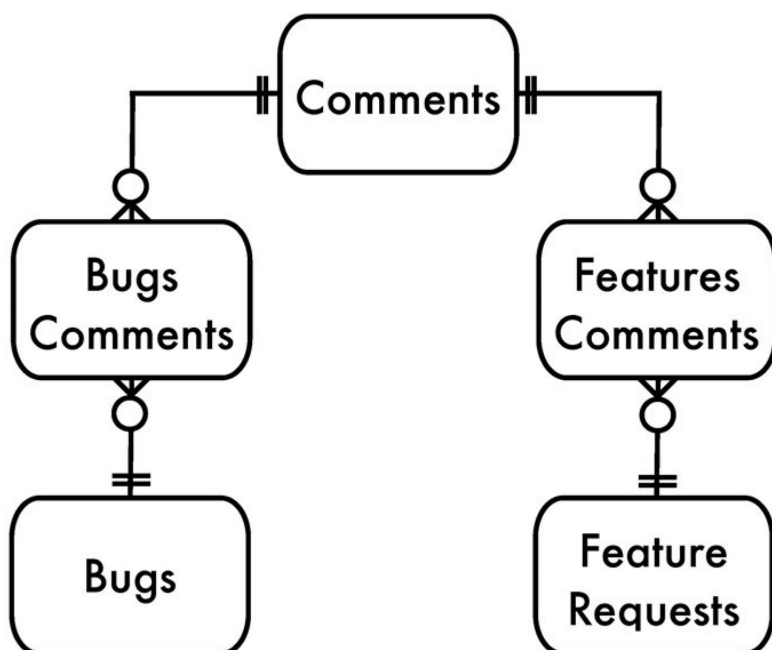


图7-4　反向多态关联

这个解决方案移除了对`Comments.issue_type`列的依赖。现在，元数据可以确保数据完整性了，从而不再依赖于应用程序代码来维护数据间的关系。

7.5.3　设立交通灯

这个方案的潜在问题是可以加入一些你可能不希望出现的关系。交叉表通常是多对多关系的模型，因而这个设计允许一个给定的评论同时和多个Bug或者多个特性记录相关。然而，你可能是希望每条评论都只涉及一个Bug或者一个特性需求。我们可以通过在每张交叉表的`comment_id`列上声明一个`UNIQUE`的约束来尽可能地支持这样的规则。

Polymorphic/soln/reverse-unique.sql

```
CREATE TABLE BugsComments (
  issue_id    BIGINT UNSIGNED NOT NULL,
  comment_id  BIGINT UNSIGNED NOT NULL,
  UNIQUE KEY (comment_id),
  PRIMARY KEY (issue_id, comment_id),
  FOREIGN KEY (issue_id) REFERENCES Bugs(issue_id),
  FOREIGN KEY (comment_id) REFERENCES Comments(comment_id)
);
```

这样就能确保一个给定的评论在一张交叉表中只能出现一次，从而确保其只能和一个Bug或者一个特性相关。然而，元数据并不能保证一个给定的评论不在多张交叉表中被关联。如果这不是你所希望的，则只能交由上层的应用程序代码来完成了。

7.5.4 双向查找

我们可以方便地使用交叉表查询一个给定的Bug或者特性需求的评论。

Polymorphic/soln/reverse-join.sql

```
SELECT *
FROM BugsComments AS b
  JOIN Comments AS c USING (comment_id)
WHERE b.issue_id = 1234;
```

我们也可以使用一个外联结查询两张交叉表来获得一条评论所对应的Bug或者特性需求记录。虽然在查询时仍不得不给出所有相关的表名，但已经比使用多态关联时简单不少。同样地，使用交叉表还能获得使用多态关联所不能提供的引用完整性支持。

Polymorphic/soln/reverse-join.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN (BugsComments JOIN Bugs AS b USING (issue_i
    USING (comment_id)
  LEFT OUTER JOIN (FeaturesComments JOIN FeatureRequests AS f
```

```
    USING (comment_id)
WHERE c.comment_id = 9876;
```

7.5.5 合并跑道

某些情况下你并没有使用之前描述的多表结构，而将多张父表都存在同一张表中（详见6.5节），你可以使用如下的两种方法来获取你所需要的结果。

首先我们来看下面这个使用UNION的查询：

Polymorphic/soln/reverse-union.sql

```
SELECT b.issue_id, b.description, b.reporter, b.priority, b.s
    b.severity, b.version_affected,
    NULL AS sponsor
  FROM Comments AS c
    JOIN (BugsComments JOIN Bugs AS b USING (issue_id))
      USING (comment_id)
  WHERE c.comment_id = 9876;
UNION
  SELECT f.issue_id, f.description, f.reporter, f.priority, f
    NULL AS severity, NULL AS version_affected,
    f.sponsor
  FROM Comments AS c
    JOIN (FeaturesComments JOIN FeatureRequests AS f USING (i
      USING (comment_id)
  WHERE c.comment_id = 9876;
```

如果你的程序已经确定将每条评论只和一张父表关联，那么这个查询是能够保证每次只返回一条记录的。由于UNION查询只有在列的数量和数据类型都一样时，才能将查询结果合并，因此你必须要为每一个父表不同的列提供一个null占位符。在UNION的查询中，你还必须用同样的顺序排列两次查询的列。

另一种方法，可以先看一下下面这个使用SQL的COALESCE()函数的查询。这个函数返回第一个非空的结果。由于我们使用了外联结查询，一条和需求相关并且在Bugs表中没有匹配项的评论，所有b.*的列都将被赋值为空。同样地，一条和Bug相关的评论，所有f.*的列都将被默认地填入空值。用简洁的方式列出针对某一父表专有的列：凡是该记录和某一父表无关，则那些字段均返回null。

Polymorphic/soln/reverse-coalesce.sql

```
SELECT c.*,
  COALESCE(b.issue_id,    f.issue_id   ) AS issue_id,
  COALESCE(b.description, f.description) AS description,
  COALESCE(b.reporter,    f.reporter   ) AS reporter,
  COALESCE(b.priority,    f.priority   ) AS priority,
  COALESCE(b.status,      f.status     ) AS status,
  b.severity,
  b.version_affected,
  f.sponsor
FROM Comments AS c
  LEFT OUTER JOIN (BugsComments JOIN Bugs AS b USING (issue_i
    USING (comment_id)
  LEFT OUTER JOIN (FeaturesComments JOIN FeatureRequests AS f
    USING (comment_id)
WHERE c.comment_id = 9876;
```

这几个查询的方案都比较复杂，因此，比较好的方式是通过它们创建数据库视图，然后就可以在你的应用程序中较为简便地使用。

7.5.6 创建共用的超级表

在面向对象的多态机制中，两个继承自同一个父类的子类型可以使用相似的方式来使用。在SQL中，多态关联这个反模式遗漏了一个关键实质：共用的父对象。我们可以通过创建一个基类表，并让所有的父表都从这个基类表扩展出来的方法来解决这个问题（参考6.5节）。在Comments子表中添加一个指向基类表的外键，并且不再需要issue_type列。这个解决方案的实体图可以用图7-5表示。

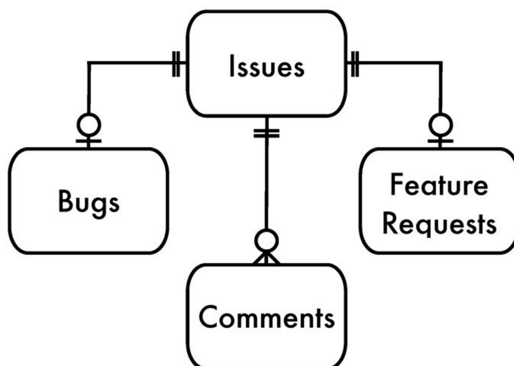


图7-5　评论和基类表的联合

Polymorphic/soln/super-table.sql

```
CREATE TABLE Issues (
  issue_id     SERIAL PRIMARY KEY
);

CREATE TABLE Bugs (
  issue_id     BIGINT UNSIGNED PRIMARY KEY,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  . . .
);

CREATE TABLE FeatureRequests (
  issue_id     BIGINT UNSIGNED PRIMARY KEY,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
. . .
);

CREATE TABLE Comments (
  comment_id   SERIAL PRIMARY KEY,
  issue_id     BIGINT UNSIGNED NOT NULL,
  author       BIGINT UNSIGNED NOT NULL,
  comment_date DATETIME,
  comment      TEXT,
  FOREIGN KEY (issue_id) REFERENCES Issues(issue_id),
  FOREIGN KEY (author) REFERENCES Accounts(account_id),
);
```

请注意Bugs和FeatureRequests表的主键也同时是外键。它们引用了Issues表所维护的代理主键，而不用自己创建它们。

给定一个评论，你可以通过一个相对简单的查询就获取对应的Bug记录或者需求记录。而不再需要在查询中包含Issues表，除非你将一些属性定义在那张表里。同样地，由于Bugs表的主键和它的祖先Issues表中的值是一样的，你可以直接对Bugs表和Comments表进行联结查询。也可以对两张没有外键约束直接关联的表进行联结查询，只要对应的列的信息是可比较的即可。

Polymorphic/soln/super-join.sql

```
SELECT *
FROM Comments AS c
  LEFT OUTER JOIN Bugs AS b USING (issue_id)
  LEFT OUTER JOIN FeatureRequests AS f USING (issue_id)
WHERE c.comment_id = 9876;
```

对于一个指定Bug，你同样可以轻易地读出它的评论。

> Polymorphic/soln/super-join.sql

```
SELECT *
FROM Bugs AS b
  JOIN Comments AS c USING (issue_id)
WHERE b.issue_id = 1234;
```

更重要的是如果你使用了像`Issues`这样的祖先表，就可以依赖外键来确保数据的完整性。

> 在每个表与表的关系中，都有一个引用表和一个被引用表。

第8章　多列属性

> 超群和荒谬的分界线往往非常模糊，很难明确地区分。
>
> 托马斯·潘恩

我已经数不清创建过多少次存储联系人信息的表了。每次都有一些共同的字段，比如名字、称呼、地址、公司等。

电话号码有一点棘手。通常人们会同时拥有多个号码：家庭电话，工作电话，传真号码以及手机号码。在联系人信息表中，分4列来存储这些信息是很简单的方法。

但对于其他号码呢？这个人的助理的号码、另一个移动电话的号码，或者外地办事处的全然不同的电话号码，甚至还有些无法预计的分类。我可以为这些并不常有的情况创建更多的列，但这看上去很笨拙，因为加了很多很少使用的字段。而且，到底要加多少这样的列才够呢？

8.1 目标：存储多值属性

这是和第2章一样的目标：一个属性看上去虽然只属于一张表，但同时可能会有多个值。之前我们已经看到，将多个值合并在一起并用逗号分隔导致难以对数据进行验证，难以读取或者改变单个值，同时也对聚合公式（诸如统计不同值的数量）非常不友好。

我们将使用一个新的例子来说明这一反模式。我们将让这个Bug数据库允许加入标签，因而就可以以此来分类Bug。某些Bug可能是由它们所影响的软件子系统，比如打印、报表或者邮件来分类的；另一些Bug可能是由它们的类型来分类的，比如一个造成程序崩溃的Bug会被标记为crash，也可以标记为performance来说明性能问题，当然还可以标记cosmetic来说明用户界面的颜色选择不好。

这个给Bug打标签的特性必须支持多标签，因为标签并不会相互排斥。一个Bug可能同时影响到多个子系统，也有可能会影响到子系统的某个特性，比如“打印的性能”。

8.2 反模式：创建多个列

我们依旧需要考虑一个属性的多个值，但我们知道每列最好只存储一个值。在这张表中创建多个列，每个列只存储一个标签看上去很自然。

Multi-Column/anti/create-table.sql

```
CREATE TABLE Bugs (
  bug_id      SERIAL PRIMARY KEY,
  description VARCHAR(1000),
  tag1        VARCHAR(20),
  tag2        VARCHAR(20),
  tag3        VARCHAR(20)
);
```

当你将一个标签指定给一个Bug记录时，必须将这个标签存放于这三个列中的一个。其他未使用的列将保持空的状态。

Multi-Column/anti/update.sql

bug_id	description	tag1	tag2	tag3
1234	Crashes while saving	crash	NULL	NULL
3456	Increase performance	printing	performance	NULL
5678	Support XML	NULL	NULL	NULL

在使用传统属性设计的时候，很简单的任务现在变得更复杂了。

8.2.1 查询数据

当根据一个给定标签查询所有Bug记录时，你必须搜索所有的三列，因为这个标签字符串可能存放于这三列中的任何一列。

比如，要获取被标记为performance的Bug，需要使用如下的一个查询表达式：

Multi-Column/anti/search.sql

```
SELECT * FROM Bugs
WHERE tag1 = 'performance'
   OR tag2 = 'performance'
   OR tag3 = 'performance';
```

你还可能需要查找同时被标记为performance和printing的Bug。要完成这样的查询，就需要写如下的查询语句。请注意必须正确地使用括号，因为OR操作比AND操作的优先级要低。

Multi-Column/anti/search-two-tags.sql

```
SELECT * FROM Bugs
WHERE (tag1 = 'performance' OR tag2 = 'performance' OR tag3 =
  AND (tag1 = 'printing' OR tag2 = 'printing' OR tag3 = 'prin
```

在多个列中查找一个值的语法是冗长乏味的。你可以通过一种非传统的方式来使用IN，从而使得这个查询变得更加精简：

Multi-Column/anti/search-two-tags.sql

```
SELECT * FROM Bugs
WHERE 'performance' IN (tag1, tag2, tag3)
  AND 'printing'    IN (tag1, tag2, tag3);
```

8.2.2 添加及删除值

在这个列的集合中添加以及删除一个值也是有问题的。单纯地使用UPDATE语句来更新一列的值是不安全的，因为你无法得知到底哪一列是有值的（如果有的话）。你可能不得不将整行数据读取到前端程序中来分析。

Multi-Column/anti/add-tag-two-step.sql

```
SELECT * FROM Bugs WHERE bug_id = 3456;
```

举个例子，假设我们知道tag2是空的。于是写出如下的UPDATE语句。

Multi-Column/anti/add-tag-two-step.sql

```
UPDATE Bugs SET tag2 = 'performance' WHERE bug_id = 3456;
```

这么做，你就要面临多步操作间的数据同步问题，即有可能在你完成一次查询并且更新记录这两步操作之间，有另一个客户端也同时在执行相同的操作。根据更新操作执行顺序的不同，你或者另一个客户端将得到一个“更新冲突”的错误，或者一方覆盖了另一方的数据。你可以使用更复杂的SQL语句来避免这样的问题。

如下的表达式使用了NULLIF()[1]函数来将每一个等于指定值的列置为空。当传入的两个参数相等时，NULLIF()函数返回空。

[1] NULLIF()在SQL中是一个标准函数，它被除Informix和Ingres以外的所有厂商支持。

Multi-Column/anti/remove-tag.sql

```
UPDATE Bugs
SET tag1 = NULLIF(tag1, 'performance'),
    tag2 = NULLIF(tag2, 'performance'),
    tag3 = NULLIF(tag3, 'performance')
WHERE bug_id = 3456;
```

如下的表达式将performance标签加到第一个空列中。然而，如果3列都不为空，这条语句将不对这条记录做任何修改，新的标签将不会被记录。同时，写这样的查询语句是非常繁琐耗时的，你必须要重复performance这个字符串6次！

Multi-Column/anti/add-tag.sql

```
UPDATE Bugs
SET tag1 = CASE
      WHEN 'performance' IN (tag2, tag3) THEN tag1
      ELSE COALESCE(tag1, 'performance') END,
    tag2 = CASE
      WHEN 'performance' IN (tag1, tag3) THEN tag2
      ELSE COALESCE(tag2, 'performance') END,
    tag3 = CASE
      WHEN 'performance' IN (tag1, tag2) THEN tag3
      ELSE COALESCE(tag3, 'performance') END
WHERE bug_id = 3456;
```

8.2.3　确保唯一性

你可能并不希望同一个值出现在多个列中，但当你使用多值属性这个反模式时，数据库并不能阻止这样的情况发生。换而言之，很难阻止如下语句的执行：

Multi-Column/anti/insert-duplicate.sql

```
INSERT INTO Bugs (description, tag1, tag2, tag3)
  VALUES ('printing is slow', 'printing', 'performance', 'per
```

8.2.4　处理不断增长的值集

这个设计的另一个弱点在于三列可能并不够用。要保证每一列只存储一个值，你必须定义和一个Bug能支持的标签最大数一样多的列。在

定义这张表的时候，你能预计多少标签数量是最大值吗？

有一个策略是暂时先猜测一个中等规模的量并在日后必要时对表进行扩展。大多数数据库允许你对已经存在的表进行重构，因此你可以增加`Bug.tag4`这个列，或者在需要时增加更多的列。

Multi-Column/anti/alter-table.sql

```
ALTER TABLE Bugs ADD COLUMN tag4 VARCHAR(20);
```

然而，这样的改变在以下三点上的开销是巨大的。

- 重构一张已经存在数据的表可能会导致锁住整张表，并阻止那些并发客户端的访问。
- 有些数据库是通过定义一张符合需求的新表，然后将现有数据从旧表中复制到新表中，再丢弃旧表的方式来实现重构表结构的。如果需要重构的表有很多数据，那转换过程将非常耗时。
- 在多列属性中增加了一列之后，你必须检查每一条相关的SQL语句，修改这些SQL语句以支持这些新加入的列。

Multi-Column/anti/search-four-columns.sql

```
SELECT * FROM Bugs
WHERE tag1 = 'performance'
  OR tag2 = 'performance'
  OR tag3 = 'performance'
  OR tag4 = 'performance'; -- you must add this new term
```

这是一个细致且费时的开发任务。如果你漏掉了任何需要修改的查询语句，就可能造成难以定位的错误。

8.3 如何识别反模式

> 反模式的模式
>
> 乱穿马路和多值属性这两个反模式都有一个共同的主线：这两个反模式都是解决同一个目标的解决方案——存储一个具有多个值的属性。

> 在乱穿马路的例子中，我们解决了多对多关系的存储。在本章中，我们将看到如何解决一对多的关系。不过需要明白的是，这两个反模式和两种数据间的关系有时是可以互用的。

如果用户界面上或者项目的设计文档中有任何属性可能具有多个值，并且这个值的数量具有一个固定的最大值，这可能意味着你使用了多值属性这个反模式。

诚然，有些属性的确可能为了某种目的，其候选值的数量具有最大值的限制，但通常情况下是没有这种限制的。如果这个限制是任意加上去的或者看上去不太合理，那可能就是因为使用了这个反模式了。

如果你听到有人说了下面这些话，那也是一个线索，提醒你可能使用了本章的反模式：

- “我们应该支持的标签数量的最大值是多少？”

  你需要决定为标签这样的多值属性定义多少列。

- “我要怎么才能在SQL查询中同时搜索多列？”

  如果你正在多列中查找一个给定的值，这是一个线索提示你应该存储多个具有同样逻辑属性的列。

8.4 合理使用反模式

在某些情况下，一个属性可能有固定数量的候选值，并且对应的存储位置和顺序都是固定的。比如，一个给定的Bug可能和多个用户账号相关，但每个关系的作用都是唯一的：一个是报告这个Bug的用户，另一个是修复这个Bug的开发人员，另一个是验证Bug修复状态的质量控制工程师。即使这几列里存储的值是相似的，它们的作用以及实际的业务逻辑都是不同的。

在`Bugs`表中定义三个不同的列来存储这三个属性是合理的。本章所描述的那些缺点在这里无关紧要，因为基本上这几列都是分开使用的。虽然有时仍旧需要对所有这三列数据进行查询，比如在一份“每个Bug都涉及哪些人”的报告里就需要这么做。既然这样的设计能够使得大部分的查询用例都变得很简单，仅在有限数量的查询用例上表现得复杂还是能够被接受的。

另一种组织数据的方式是创建一张从属表来存储Bugs表和Accounts表之间的关系，同时在这张额外的表中增加一列来记录一个关系中的账号所表示的角色。然而，这样的设计可能会导致第6章EAV所描述的一些问题。

8.5 解决方案：创建从属表

如同我们在第2章中所看到的那样，最好的解决方案是创建一张从属表，仅使用一列来存储多值属性。将多个值存在多行中而不是多列里。同时，在从属表中定义一个外键，将这个值和Bugs表中的主记录关联起来。

Multi-Column/soln/create-table.sql

```
CREATE TABLE Tags (
  bug_id        BIGINT UNSIGNED NOT NULL
  tag           VARCHAR(20),
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);

INSERT INTO Tags (bug_id, tag)
  VALUES (1234, 'crash'), (3456, 'printing'), (3456, 'perform
```

当所有和Bug相关的标签都存储于一列中时，查找和一个给定标签相关的所有Bug就变得很直接了。

Multi-Column/soln/search.sql

```
SELECT * FROM Bugs JOIN Tags USING (bug_id)
WHERE tag = 'performance';
```

即使更复杂一点的查询，诸如“和两个标签相关的Bug”，其查询代码也非常简单易读。

Multi-Column/soln/search-two-tags.sql

```
SELECT * FROM Bugs
  JOIN Tags AS t1 USING (bug_id)
```

```
  JOIN Tags AS t2 USING (bug_id)
WHERE t1.tag = 'printing' AND t2.tag = 'performance';
```

你可以使用比多值属性反模式更简单的方法来添加或移除数据间的关系——只要简单地添加或者删除从属表中的记录就行了。不再需要逐列检查是否有空列可以添加记录了。

> Multi-Column/soln/insert-delete.sql

```
INSERT INTO Tags (bug_id, tag) VALUES (1234, 'save');

DELETE FROM Tags WHERE bug_id = 1234 AND tag = 'crash';
```

主键的约束能够保证不会有重复记录出现。一个给定的标签只能和一个给定的Bug关联一次，如果尝试重复插入，SQL会告诉你错误。

每个Bug不再限制只能打三个标签了，不像在Bugs表中只能加三个tagN的列，现在只要有需要就可以一直加新的标签。

> 将具有同样意义的值存在同一列中。

第9章　元数据分裂

> 我不想再在船上看到这些东西。即使要付出我们所有的人作为代价我也不管，我就是要它们下船。
>
> 詹姆斯·柯克

我的妻子当Java及Oracle PL/SQL程序员已经好几年了。她提供了一个案例来展示什么样的数据库设计能使得工作变得更简单。

她的公司的销售部门使用的数据库中有一张叫做Customers的表，记录了和客户相关的信息，比如客户的业务类型，以及已经从这个客户那里获得了多少收入。

> Metadata-Tribbles/intro/create-table.sql

```
CREATE TABLE Customers (
  customer_id   NUMBER(9) PRIMARY KEY,
  contact_info  VARCHAR(255),
  business_type VARCHAR(20),
  revenue       NUMBER(9,2)
);
```

但是销售部门需要按年来划分这些收入以便于跟踪每个客户的动态。他们决定增加一系列的列，每一列都按照年份来命名。

Metadata-Tribbles/intro/alter-table.sql

```
ALTER TABLE Customers ADD (revenue2002 NUMBER(9,2));
ALTER TABLE Customers ADD (revenue2003 NUMBER(9,2));
ALTER TABLE Customers ADD (revenue2004 NUMBER(9,2));
```

接着他们输入不完整的数据，只包含那些他们有兴趣跟踪的客户。大多数的行中，他们将这些收入字段置空。开发人员怀疑他们会不会在这些几乎用不到的列中存些别的数据？

每一年，他们都往这张表中增加一个新列。有一个专门的数据库管理员负责维护Oracle的表空间。因此，每一年，他们都需要开一系列的会议，协调数据迁移和重新组织表空间，并且增加新列。最终，他们浪费了一堆时间和金钱。

9.1 目标：支持可扩展性

随着数据量的增长，数据库的查询性能也会随之下降。即使查询结果可能只有很少的几千行，遍历表中积累的数据也可能使得整个查询的性能变得极差。即使使用了索引，但随着数据的增长，索引的作用变得非常有限。

本章的目标就是要优化数据库的结构来提升查询的性能以及支持表的平滑扩展。

9.2 反模式：克隆表与克隆列

在“星际迷航”中，tribbles是一种被当做宠物饲养的小巧的、毛茸茸的动物。tribbles最开始非常地吸引人，但很快它们就暴露出本性，开始

了无节制的繁殖，然后管理泛滥的tribbles成了船上的一个严重问题。

你该把这些tribbles放哪儿？谁该为这些tribbles负责？要将所有的tribbles都收集起来要花多长时间？最终，柯克船长发现他的飞船及船员们无法正常工作，他不得不下令将“赶走tribbles”作为头等大事来处理。

根据经验，我们知道查询一张表时，其性能只和这张表中数据的条数相关，越少的记录，查询速度越快。于是我们推导出一个常见错误的结论：无论要做什么，我们必须让每张表存储的记录尽可能少。这就导致了本章的反模式的两种表现形式。

- 将一张很长的表拆分成多张较小的表，使用表中某一个特定的数据字段来给这些拆分出来的表命名。
- 将一个列拆分成多个子列，使用别的列中的不同值给拆分出来的列命名。

但是你不能不劳而获：为了要达成减少每张表记录数的目的，你不得不创建一些有很多列的表，或者创建很多很多表。但在这两个方案中，你会发现随着数据量的增长，会有越来越多的表或者列，因为新的数据迫使你创建新的Schema对象。

> 混淆元数据和数据
>
> 请注意，通过将年份追加在基本表名之后，我们其实是将数据和元数据标识合并在了一起。
>
> 这和我们早先在EAV和多态关联反模式中看到的混合数据和元数据的方式正好相反。在那些案例中，我们将元数据标识（列名和表名）当做字符串存储。
>
> 在多列属性和元数据分裂模式中，我们将数据的值存储在列名或者表名中。如果你使用任何这些反模式，你只会得到更多的问题。

9.2.1 不断产生的新表

要将数据拆分到不同的表中，需要一个规则来定义哪些数据属于哪些表。比如，可以根据`date_reported`中的年份进行拆分：

Metadata-Tribbles/anti/create-tables.sql

```
CREATE TABLE Bugs_2008 ( . . . );
CREATE TABLE Bugs_2009 ( . . . );
CREATE TABLE Bugs_2010 ( . . . );
```

由于要将数据添加进数据库，于是根据要添加的数据的值选择合适的表就成了你的责任：

Metadata-Tribbles/anti/insert.sql

```
INSERT INTO Bugs_2010 (..., date_reported, ...) VALUES (...,
```

让我们快进到2011年1月1日。你的程序在添加新的Bug报告时不断的报错，因为你忘记添加一张叫做`Bugs_2011`的新表。

Metadata-Tribbles/anti/insert.sql

```
INSERT INTO Bugs_2011 (..., date_reported, ...) VALUES (...,
```

这意味着新的数据可能会需要新的元数据对象。这在SQL的设计中并不是常见的数据与元数据的关系。

9.2.2 管理数据完整性

假设你的老板打算计算这一年中Bug的数量，但他得到的数字并不正确。检查了相关数据和程序之后，你发现有一部分2010年的Bug被误写入了`Bugs_2009`这张表。如下的查询语句应该每次都返回空结果，如果不是的话，那么就有麻烦了：

Metadata-Tribbles/anti/data-integrity.sql

```
SELECT * FROM Bugs_2009
WHERE date_reported NOT BETWEEN '2009-01-01' AND '2009-12-31'
```

没有任何办法自动地对数据和相关表名做限制，但可以在每张表中都声明一个`CHECK`的约束：

Metadata-Tribbles/anti/check-constraint.sql

```
CREATE TABLE Bugs_2009 (
  -- other columns
  date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported) =
);
CREATE TABLE Bugs_2010 (
  -- other columns
  date_reported DATE CHECK (EXTRACT(YEAR FROM date_reported) =
);
```

注意当你创建Bugs_2011时要记得修改CHECK约束里的值。如果你犯错了，就可能创建出一张拒绝接受正确值的表。

9.2.3 同步数据

某一天，你的客户支持部分析师想要改变Bug的日期。在数据库中存储的日期是2010-01-03，但顾客实际是在一周以前，即2009-12-27，使用传真报告的错误。你可以使用一个简单的UPDATE语句来修改这个值：

Metadata-Tribbles/anti/anomaly.sql

```
UPDATE Bugs_2010
SET date_reported = '2009-12-27'
WHERE bug_id = 1234;
```

但这个修正使得Bugs_2010表中的某一条记录变成了无效记录。你需要先从这张表中移除这条记录然后插入到另一张表中，在少数情况下，一个简单的UPDATE语句就会造成这样的异常状况。

Metadata-Tribbles/anti/synchronize.sql

```
INSERT INTO Bugs_2009 (bug_id, date_reported, ...)
  SELECT bug_id, date_reported, ...
  FROM Bugs_2010
  WHERE bug_id = 1234;

DELETE FROM Bugs_2010 WHERE bug_id = 1234;
```

9.2.4 确保唯一性

需要确保所有被分割出来的表中的主键都是唯一的。如果你需要从一张表中移动一条记录到另一张表中，需要保证被移动记录的主键值不会和目标表中的主键记录冲突。

如果使用一个支持序列化对象的数据库，那么可以使用一个简单的序列来确保主键值在所有的分割表中都是唯一的。对于那些只支持单表ID唯一的数据库产品来说，实现这样的功能变得很不优雅。你不得不定义一张额外的表来存储产品主键的值：

Metadata-Tribbles/anti/id-generator.sql

```
CREATE TABLE BugsIdGenerator (bug_id SERIAL PRIMARY KEY);

INSERT INTO BugsIdGenerator (bug_id) VALUES (DEFAULT);
ROLLBACK;

INSERT INTO Bugs_2010 (bug_id, . . .)
  VALUES (LAST_INSERT_ID(), . . .);
```

9.2.5 跨表查询

不可避免地，你的老板肯定会需要查询多张表中的数据。比如，他可能会要求查询所有的Bug总数，不管是哪一年报告的。你可以使用UNION将所有分割表联合起来得到一个重构过的Bug集合并将其作为一个衍生表进行查询：

Metadata-Tribbles/anti/union.sql

```
SELECT b.status, COUNT(*) AS count_per_status FROM (
  SELECT * FROM Bugs_2008
    UNION
  SELECT * FROM Bugs_2009
    UNION
  SELECT * FROM Bugs_2010 ) AS b
GROUP BY b.status;
```

年复一年，你创建了越来越多的表，比如Bugs_2011，你需要不断地更新程序代码来引入这些新创建的表。

9.2.6 同步元数据

你的老板要求你在表中增加一列用以记录解决每个Bug所用的时间。

Metadata-Tribbles/anti/alter-table.sql

```
ALTER TABLE Bugs_2010 ADD COLUMN hours NUMERIC(9,2);
```

如果将表进行了拆分，那么这个新列仅仅被应用到你选择的那张表上。其他所有的表都不包含这个新列。

如果使用前一段描述的那个UNION查询所有的分割表，就会碰到一个新问题：当多张表具有相同的列时才能使用UNION。如果多张表的列不完全相同，你就必须指明所有表都同时拥有的列，而不可以再使用“*”通配符。

9.2.7 管理引用完整性

如果一个关联表，比如Comments引用了Bugs，这个关联表就不能声明一个外键了。一个外键必须指定单个表，但在这个案例中父表被拆分成很多个表。

Metadata-Tribbles/anti/foreign-key.sql

```
CREATE TABLE Comments (
  comment_id         SERIAL PRIMARY KEY,
  bug_id             BIGINT UNSIGNED NOT NULL,
  FOREIGN KEY (bug_id) REFERENCES Bugs_????(bug_id)
);
```

分割表即使作为一张关联表而不是父表，同样可能引起一些问题。比如，Bugs.reported_by引用了Accounts表。如果你想要查询一个给定的人所报告的所有Bug，不管哪一年的，你需要使用像下面的这个查询：

Metadata-Tribbles/anti/join-union.sql

```
SELECT * FROM Accounts a
JOIN (
```

```
  SELECT * FROM Bugs_2008
  UNION ALL
  SELECT * FROM Bugs_2009
  UNION ALL
  SELECT * FROM Bugs_2010
) t ON (a.account_id = t.reported_by)
```

9.2.8　标识元数据分裂列

列也可能根据元数据分裂。你可以创建一个含有很多列的表，这些列按照它们的类别扩展，就像我们在本章最开始看到的那个故事一样。

我们的Bug数据库中可能碰到的另一个事例是：有一张表记录了项目指标的概要信息，其中的每一列存储了一个小计。具体来说，在如下这张表中，增加一个`bugs_fixed_2011`列只是时间问题：

Metadata-Tribbles/anti/multi-column.sql

```
CREATE TABLE ProjectHistory (
  bugs_fixed_2008  INT,
  bugs_fixed_2009  INT,
  bugs_fixed_2010  INT
);
```

9.3　如何识别反模式

如下的描述可能就是元数据分裂反模式在你的数据库中繁衍生长的暗示。

- “那么我们需要每……创建一张表（或者列）。”

  当你这样描述数据库时，说明你正根据某一列的值的范围拆分表。

- “数据库所支持的最大数量的表（或者列）是多少？”

  如果你的设计合理，大多数的数据库可以处理的表和列的数量比你所需要的多得多。如果你认为你的数据库设计可能会超过最大值，很明显应该重新考虑设计了。

- “我们终于发现为什么今早程序添加新记录失败了：我们忘记为新的一年添加新表了。”

  这是元数据分裂的普遍结果。当新的数据需要新的数据库对象时，你需要预先定义这些对象，否则就要接受这些不可预计的错误。

- “我要怎样同时查询很多张表？每张表的列都是一样的。”

  如果你需要查询很多结构一样的表，就应该将数据全都存在一个表中，使用一个额外的属性列来分组数据。

- “我要怎样将表名作为一个变量传递？我在查询时需要根据年份动态地生成这些表名。”

  如果你的数据都在一张表里，那你根本不需要做这些事情。

9.4 合理使用反模式

手动分割表的一个合理使用场景是归档数据——将历史数据从日常使用的数据中移除。通常在过期数据的查询变得非常稀少的情况下，才会进行如此的操作。

如果你没有同时查询当前数据和历史数据的需求，将老数据从当前活动的表转移到其他地方是很合适的操作。

将数据归档到和当前表结构相兼容的新表中，既能支持偶尔做数据分析时的查询，同时也能让日常数据查询变得非常高效。

> **WordPress.com使用的分区数据库设计**
>
> 在2009年的MySQL大会上，我和Barry Abrahamson一起吃饭，他是WordPress.com的数据库架构师，而WordPress.com是一个很流行的博客服务。
>
> Barry说他开始做博客主机服务时，将所有客户的数据存在一个数据库中，毕竟每个博客站点的数据量并不是很大。在当时，单个数据库便于管理是个很好的理由。
>
> 网站最初建立的时期，这样的设计工作得很好，但很快就发展成大规模的数据库操作。现在他们在300台数据库服务器上存储7百

> 万博客数据。每台服务器为一部分用户服务。
>
> Barry添加新的服务器时，对存储着所有用户博客信息的单一数据库进行拆分是非常痛苦的事情。通过将每个用户的数据分开存储到单独的数据库中，Barry发现将一个用户的数据从一台服务器转移到另一台服务器变得异常简单。由于用户总是在不断地流动，有些用户的博客流量非常高，而另一些用户的博客则相对比较冷清，Barry所负责的不断调整服务器之间的负载均衡工作就变得很重要。
>
> 备份并恢复一个中等规模的数据库比操作一个存储着TB级数据的数据库要方便得多。比如，如果一个用户打电话来说由于输入了错误的数据，他们的数据现在变得一团糟，Barry该怎么恢复这个用户的数据，如果所有用户的数据都存于同一个巨大的数据库中？
>
> 尽管将数据对象模型化并将整个对象中的所有东西映射到一个单独的数据库中的做法没有错，但合理地将大小超过临界值的数据库拆分开能简化数据库管理的工作。

9.5　解决方案：分区及标准化

当一张表的数据量变得非常巨大时，除了手动拆分这张表，还有更好的办法来提升查询性能。这些方法就包括了水平分区、垂直分区以及使用关联表。

9.5.1　使用水平分区

你可以使用一种称为水平分区或者分片的数据库特性来分割大数据量的表，同时又不用担心那些分割表所带来的缺陷。你仅需要定义一些规则来拆分一张逻辑表，数据库会为你管理余下的所有事情。物理上来说，表的确是被拆分了，但你依旧可以像查询单一表那样执行SQL查询语句。

定义每个表拆分的方式是非常灵活的。比如，使用MySQL5.1所支持的分区特性，你可以在`CREATE TABLE`语句中将分区规则作为可选参数。

> Metadata-Tribbles/soln/horiz-partition.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  -- other columns
  date_reported DATE
) PARTITION BY HASH ( YEAR(date_reported) )
  PARTITIONS 4;
```

上例中分割数据库的方式和这章最开始讲到的方法类似，根据date_reported列里的年份对数据进行拆分。然而，这么做的优势在于，相比于手动拆分表，你永远不用担心数据会放入错误的分割表中，即使date_reported列的值更新了。而且，你不必引用所有的分割表就能对Bugs表进行整体的查询操作。

实际存储数据的物理表在本例中被固定设置为4张。当记录的年份跨度超过4年，某一个分区将用来存储多于一年的数据。年份跨度不断增长，这样的现象也会不断重演。你不必添加新的分区，除非分区里的数据量变得非常巨大，让你觉得需要重新分区。

分区在SQL标准中并没有定义，因此每个不同的数据库实现这一功能的方式都是非标准的。对应的术语、语法和明确的特性定义在不同的数据库中有非常大的区别。然而，某些形式的分区如今已经被很大一部分数据库所支持了。

9.5.2　使用垂直分区

鉴于水平分区是根据行来对表进行拆分的，垂直分区就是根据列来对表进行拆分。当某些列非常庞大或者很少使用的时候，对表进行按列拆分会比较有优势。

BLOB类型和TEXT类型的列的大小是可变的，可能非常大。为了提高存储和查询的性能，有些数据库自动地将这些类型的列和表中其他的列分开进行存储。如果你进行一个不包含BLOB或者TEXT类型的查询，就可以更高效地获取其他的列。但如果使用一个通配符“*”来进行查询，数据库会返回这张表中所包含的所有列，包括那些类型为BLOB或者TEXT的列。

比如说，在缺陷数据库中，我们可能会在Products表中为每个单独的产品存储一份安装文件。这种文件都是自解压的，在Windows上的后缀通常为 .exe，在Mac上后缀为 .dmg。这种文件通常都很大，但BLOB类型的列可以存储庞大的二进制数据。

从逻辑上讲，安装文件是Products表的一个属性，但在绝大多数针对这张表的查询中，安装程序通常是不需要的。如果你有使用通配符“*”进行查询的习惯，那么将如此大的文件存储在Products表中，而且又不经常使用，很容易就会在查询时遗漏这一点，从而造成不必要的性能问题。

正确的做法是将BLOB列存在另一张表中，和Products表分离但又与其相关联。让这张BLOB表的主键同时作为一个指向Products表的外键，用以确保每个产品最多有一条与之对应的安装包记录。

Metadata-Tribbles/soln/vert-partition.sql

```
CREATE TABLE ProductInstallers (
  product_id      BIGINT UNSIGNED PRIMARY KEY,
  installer_image BLOB,
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

之前的例子虽然比较极端，但它确实展示了将一些列从某一张表中分离出来的优势。比如，在MySQL的MyISAM存储引擎中，对一个所有行的大小都是固定的表是最高效的。VARCHAR是一个可变长数据类型，因此，只要表中出现一个这样类型的字段，那就无法享受固定长度的查询速度优势。如果你将所有可变长字段都存储在分离的表中，对主表的查询效率就能有所提高（即使只有一点点）。

Metadata-Tribbles/soln/separate-fixed-length.sql

```
CREATE TABLE Bugs (
  bug_id        SERIAL PRIMARY KEY, -- fixed length data type
  summary       CHAR(80),           -- fixed length data type
  date_reported DATE,               -- fixed length data type
  reported_by   BIGINT UNSIGNED,    -- fixed length data type
  FOREIGN KEY  (reported_by) REFERENCES Accounts(account_id)
);

CREATE TABLE BugDescriptions (
  bug_id       BIGINT UNSIGNED PRIMARY KEY,
  description  VARCHAR(1000),       -- variable length data ty
  resolution   VARCHAR(1000)        -- variable length data ty
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

9.5.3 解决元数据分裂列

和我们在第8章多列属性中使用的解决方案类似，解决元数据分裂的改进方案就是创建关联表。

> Metadata-Tribbles/soln/create-history-table.sql

```
CREATE TABLE ProjectHistory (
  project_id  BIGINT,
  year        SMALLINT,
  bugs_fixed  INT,
  PRIMARY KEY (project_id, year),
  FOREIGN KEY (project_id) REFERENCES Projects(project_id)
);
```

使用每行一个项目、每一列记录一年的Bug修复数量，还不如使用多行、仅用一列记录修复的Bug数量。如果你这样定义表，就不需要为随后的年份增加新列，随着时间的增长，你可以为每个项目存储任意数量的记录。

> 别让数据繁衍元数据。

第二部分　物理数据库设计反模式

本部分内容

- 第10章　取整错误
- 第11章　每日新花样
- 第12章　幽灵文件
- 第13章　乱用索引

第10章　取整错误

> **10.0乘以0.1未必就是1.0。**
>
> 布莱思·克尼汉

老板要求你根据每个程序员修复每个Bug所花费的时间，来计算并给出一个关于项目开发时间成本的报表。每个在Accounts表中的程序员都有不同的时薪，因此你统计出每个程序员花在修复每个Bug上的时间，并用其乘以对应的时薪。

> Rounding-Errors/intro/cost-per-bug.sql

```
SELECT b.bug_id, b.hours * a.hourly_rate AS cost_per_bug
FROM Bugs AS b
  JOIN Accounts AS a ON (b.assigned_to = a.account_id);
```

要实现这样的查询功能，你需要在Bugs和Accounts表中创建新的列。这些新的列都应该支持浮点数，因为你需要精确地统计这些开销。你决定将这些列的类型定义为FLOAT，因为这种类型支持浮点数。

> Rounding-Errors/intro/float-columns.sql

```
ALTER TABLE Bugs ADD COLUMN hours FLOAT;
```

```
ALTER TABLE Accounts ADD COLUMN hourly_rate FLOAT;
```

通过分析与Bug相关的工作日志及程序员的时薪，你更新了这些新列，测试了相关数据，然后结束了当天的工作。

第二天，你的老板拿了份项目开销报表出现在你的办公室里。“这些数字不正确，”他咬牙切齿地说道：“我自己手动算过一次，而你的报告是错的——虽然很小，只差几美元。你怎么解释这个问题？”你开始冒冷汗，这样简单的计算哪里会出问题呢？

10.1　目标：使用小数取代整数

整型是一个很有用的数据类型，但只能存储整数，比如1、327或者-19之类的。它不能表述像2.5这样的浮点数。如果数据对精度要求很高，你需要使用另一种数据类型来取代整型。比如，算钱的时候通常需要精确到小数点后两位，像$19.95这样。

因此，本章的目标就是存储非整数类型的数字，并且在基本运算中使用它们。还有另一个目标，运算的结果必须正确。当然，这个目标是最基本的要求，实现它是理所当然的。

10.2　反模式：使用**FLOAT**类型

大多数编程语言都支持实数类型，使用关键字`float`或者`double`。SQL也使用相同的关键字支持类似的数据类型。很多程序员很自然地就会在需要使用浮点数的地方使用SQL `FLOAT`类型，因为他们习惯于使用`float`类型编程。

SQL中的`FLOAT`类型，就和其他大多数编程语言的`float`类型一样，根据IEEE 754标准使用二进制格式编码实数数据。你需要了解一些浮点数的定义规范，才能有效地使用这个数据类型。

10.2.1　舍入的必要性

很多程序员并不清楚浮点类型的特性：并不是所有在十进制中描述的信息都能使用二进制存储。出于一些必要的因素，浮点数通常会舍入到一个非常接近的值。

让我们更加直观地了解这个取整操作。举例来说，1/3用一个无限循

环的十进制可以表示为0.333…，真实的值无法完整地写出来，因为需要写无限多个3。小数点后数字的个数表示了这个数字的精确度，因此，无限循环地写下3，能够无限接近于1/3的精确值。

折中的办法是限制精度，选择一个尽可能接近原始值的数字，比如0.333。然而，这也意味着这个数字不是我们所希望的那个值。

$$\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1.000$$
$$0.333 + 0.333 + 0.333 = 0.999$$

即使提高了精度，仍旧不能将这三个近似值加起来得到1.0。这是使用有限精度的数表示无限小数时的必要妥协。

$$\frac{1}{3} + \frac{1}{3} + \frac{1}{3} = 1.000000$$
$$0.333333 + 0.333333 + 0.333333 = 0.999999$$

这意味着，你能想到的某些合理的数是无法用有限精度表示的。你可能觉得这样问题不大，因为你确实不可能输入一个无限小数，因而认为，既然实际输入的值是有限精度的，那么也应该能以相同的精度存储？但很不幸，事实并非如此。

IEEE 754使用二进制表示浮点数。十进制中的无限小数在二进制中的表达方式是完全不同的。然而一些十进制有限小数，比如59.95，在二进制中却需要表示为无限小数。`FLOAT`类型无法表达无限小数，因而，它存储了二进制表示中最接近59.95的值，用十进制表示等于59.950000762939。

有些值碰巧在两种表达方式中能达到相同的精度，从理论上来说，如果你了解IEEE 754标准的细节，就可以预测一个给定的十进制数如何以二进制形式存储。但在实际中，大多数的人不会关心每一个他们正在使用的浮点数的实际精度。你无法保证一个`FLOAT`类型的列中存储的值都是符合精度需求的，因此你的程序应该认为这一列中的每个值都是经过舍入的。

有些数据库支持其他相类似的数据类型`DOUBLE PRECISION`和`REAL`。包括`FLOAT`在内的三种数据类型的理论精度对于不同的数据库实现来说不尽相同，但它们都使用有限个二进制位来存储浮点数，因此它们都有着相似的取整行为。

10.2.2 在SQL中使用FLOAT

有些数据库能够通过某种方式弥补数据的不精确性，输出我们所期望的值。

> **IEEE 754标准简介**
>
> 浮点数二进制表达的规范的提案可以追溯到1979年，最终是在1985年定稿。如今已经广泛地被各种程序、编程语言及微处理器所实现。
>
> 这个规范使用三段编码：一段用来表示一个值的小数部分，一段用来表示其偏置指数，剩余的另外一位表示符号。在科学计算中，IEEE 754标准的使用非常广泛，它同时能用来描述极大值以及极小值。这个标准不仅仅支持实数，其取值范围比固定大小的整型还要大。双精度浮点类型所支持的取值范围更大。因此，对于科学计算类的程序来说，这几个类型是非常有用的。
>
> 但是大多数使用浮点数的场合是用来算钱。对于钱的计算来说并不需要使用IEEE 754标准。因为本章所介绍的刻度化十进制格式能非常简单且精确地处理与钱相关的操作。
>
> 参考维基百科的文章（ ）或者David Goldberg的文章“What Every Computer Scientist Should Know About Floating-Point Arithmetic”[Gol91] 能更好地了解这个标准。
>
> Goldberg的文章可以查看http://www.validlab.com/goldberg/paper.pdf。

> Rounding-Errors/anti/select-rate.sql

```
SELECT hourly_rate FROM Accounts WHERE account_id = 123;

Returns: 59.95
```

但FLOAT类型的列中实际存储的数据可能并不完全等于它的值。如果将这个值扩大十亿倍，就能看到其中的区别：

Rounding-Errors/anti/magnify-rate.sql

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE account_i

Returns: 59950000762.939
```

你可能希望上面那个扩大的查询返回的结果应该为59950000000.000。这说明IEEE 754标准的二进制模式将59.95转化成了一个可以用有限精度表示的值。在这个例子中，取整后的值与原值的误差为千万分之一以内，对于大多数的运算来说已经足够精确了。

然而，对于某些运算来说这样的误差还是不可容忍的的。最简单的例子就是用`FLOAT`进行比较操作。

Rounding-Errors/anti/inexact.sql

```
SELECT * FROM Accounts WHERE hourly_rate = 59.95;

Result: empty set; no rows match.
```

我们知道在`hourly_rate`列中实际存储的值比59.95要稍微大一点点。即使你给`account_id`为123的`hourly_rate`赋值为59.95，之前的那个查询也会以失败而告终。

通常的变通方案是将浮点数视作“近似相等”，即两个值之间的差值足够小就认为它们相等。我们将两个值相减，并使用SQL中的`ABS()`函数取其绝对值。如果结果为0，则表示两个数绝对相等。如果结果足够小，那我们就认为这两个数是近似相等的。下面的这个查询能够返回正确的结果：

Rounding-Errors/anti/threshold.sql

```
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.000
```

然而，即使是如此细小的差值在对精度要求较高的比较中也会失败：

Rounding-Errors/anti/threshold.sql

```
```

```
SELECT * FROM Accounts WHERE ABS(hourly_rate - 59.95) < 0.000
```

由于十进制数和取整后的二进制数的绝对值不相同，我们需要合适的阈值。

另一个由于使用非精确的FLOAT造成误差的情况，是使用合计函数计算很多值的时候。比如，如果使用SUM()函数计算一列中的所有值，那最终得到的总和会受到这一列中所有非精确值的影响。

Rounding-Errors/anti/cumulative.sql

```
SELECT SUM( b.hours * a.hourly_rate ) AS project_cost
FROM Bugs AS b
JOIN Accounts AS a ON (b.assigned_to = a.account_id);
```

非精确浮点数所积累的影响对于求和之外的合计运算来说会更大。虽然误差看起来非常小，但其累加起来的效果不可忽视。比如，如果你用1精确地乘以1.0，那无论执行多少次，结果总是1。然而，如果这个乘数因子实际上是0.999，结果就完全不同。如果你用1乘以0.999一千次，你得到的结果将约等于0.3677。这样的操作执行次数越多，误差就越大。

实际中需要连续多次进行浮点数乘法运算的例子，是在金融项目中计算复利。使用非精确浮点数所造成的误差看起来很小，但会不断累加。因此，在金融项目中使用精确值是非常重要的。

10.3　如何识别反模式

实际上，任何使用FLOAT、REAL或者DOUBLE PRECISION类型的设计都有可能是反模式。大多数应用程序使用的浮点数的取值范围并不需要达到IEEE 754标准所定义的最大/最小区间。

似乎在SQL中使用FLOAT类型是很自然的事情，毕竟它和大多数编程语言中的浮点类型所使用的关键字是一样的。但对于浮点数的存储，我们还有更好的选择。

10.4　合理使用反模式

当你需要存储的数据的取值范围很大，大于INTEGER和NUMERIC这

两个类型所支持的范围时，FLOAT就是你的选择。科学计算类的程序就是FLOAT通常的应用场合。

Oracle使用FLOAT类型表示的是一个精确值，而BINARY_FLOAT类型是一个非精确值，使用的是IEEE 754标准编码。

10.5 解决方案：使用NUMERIC类型

使用SQL中的NUMERIC或DECIMAL类型来代替FLOAT及与其类似的数据类型进行固定精度的小数存储。

> Rounding-Errors/soln/numeric-columns.sql

```
ALTER TABLE Bugs ADD COLUMN hours NUMERIC(9,2);

ALTER TABLE Accounts ADD COLUMN hourly_rate NUMERIC(9,2);
```

这些数据类型精确地根据你定义这一列时指定的精度来存储数据。通过类似于指定VARCHAR的长度的方式，将精度作为类型参数来定义列的类型。其精度所指的是，在这一列中的每个值最多所能包含的有效数字的个数。精度为9意味着你可以存储123456789，而1234567890[1]则为非法值。

> [1] 在一些厂商的数据库中，列的大小内部取整到最接近的字节、字或是双字，所以NUMERIC列的最大值可能有比你规定的要多的数位。

你也可以使用该类型的第二参数来指定其刻度。这里的刻度即指小数点后的位数。小数部分的数字也算在其有效位中，因此，精度9刻度2意味着可以存储1234567.89，而12345678.91或者123456.789都是非法值。

应用在某一列上的精度和刻度会影响到这一列中的每一行记录，也就是说，你无法在某些行上存储刻度为2的记录，同时又在另一些行上存储刻度为4的记录。在SQL中一列的数据类型对于这一列中的每条记录都是通用的（就如同定义了VARCHAR(20)意味着每一行都只能存储最大长度为20的字符串）。

NUMERIC和DECIMAL的优势之处在于，它们不会像FLOAT类型那样

对存储的有理数进行舍入操作。假设你输入59.95，就可以确信实际存储的数据就是59.95。当使用存储的数据与原始数据59.95进行比较时，必然返回两者相等。

> Rounding-Errors/soln/exact.sql

```
SELECT hourly_rate FROM Accounts WHERE hourly_rate = 59.95;

Returns: 59.95
```

同样地，如果你按比例将值扩展十亿倍，就可以得到所期望的值：

> Rounding-Errors/soln/magnify-rate-exact.sql

```
SELECT hourly_rate * 1000000000 FROM Accounts WHERE hourly_ra

Returns: 59950000000
```

NUMERIC和DECIMAL这两个类型的行为是一样的，两者没有任何区别。DEC也是DECIMAL的简称。

你仍旧无法存储无限精度的数据，诸如1/3，但至少我们对十进制数的这些约束有了更深的了解。

如果你需要精确地表示十进制数，使用NUMERIC类型。FLOAT类型无法表示很多十进制的有理数，因此它们应该当成非精确值来处理。

> 尽可能不要使用浮点数。

第11章　每日新花样

> 当变量有限且可以一一列举，当变量间的组合是不重复且清晰的，那就存在科学。
>
> 保罗·瓦勒里

在个人信息表中，称呼（salutation）列可以只有有限的几个候选

值。基本上你只需要支持Mr.、Mrs.、Ms.、Dr.以及Rev.，这些称谓几乎覆盖了所有人。你可以在声明这个列的时候使用数据类型或者约束来指定这些候选值，因此不会有人往salutation列中加入无效值。

31-Flavors/intro/create-table.sql

```
CREATE TABLE PersonalContacts (
  -- other columns
  salutation VARCHAR(4)
    CHECK (salutation IN ('Mr.', 'Mrs.', 'Ms.', 'Dr.', 'Rev.'
);
```

这一列基本上可以保持稳定，因为你不需要支持别的称呼了，是这样的吗？

遗憾的是，你的老板告诉你公司正准备在法国创建一家分公司。你需要支持M.、Mme.以及Mlle.这些称呼。你的任务就是让你的联系人表能接受这些值。这是一项很精细的任务，不中断表的读写操作可能无法完成这一任务。

你同时还得考虑，你的老板提到了下月可能会在巴西建立办事处的事情。

11.1 目标：限定列的有效值

将一列的有效字段值约束在一个固定的集合内是非常有用处的。如果我们可以确保一列中永远不会包含无效字段，那对于使用方来说，逻辑会变得非常简单。

比如在Bugs表中，status列标记了一个给定的Bug是NEW、IN PROGRESS、FIXED等状态。这些值的意义与在项目中如何管理这些Bug有关，但我们现在所关心的是status这一列中的值必须限定在所列出来的这几个值中。

理想情况下，我们希望数据库能够拒绝无效值的输入：

31-Flavors/obj/insert-invalid.sql

```
INSERT INTO Bugs (status) VALUES ('NEW'); -- OK
```

```
INSERT INTO Bugs (status) VALUES ('BANANA'); -- Error!
```

> 芭斯罗缤的**31**种冰激凌
>
> 从1953年开始，著名的冰激凌连锁店芭斯罗缤（Baskin-Robbins）在一个月中每天提供一种不同的口味。他们使用“31 Flavors”这个口号很多年。
>
> 如今，60多年后，芭斯罗缤提供21种经典口味，12种季节性口味，16种地区性口味，另外还有各种各样的Bright Choices和Flavors of the Month。以前芭斯罗缤的冰激凌口味根据他们的招牌是固定不变的，但他们后来对此做了扩展，可配置，可变化。在你设计数据库时，也可以使用同样的方式——实际上，你确实应该使用这种方式。

11.2　反模式：在列定义上指定可选值

很多数据库设计人员趋向于在定义列的时候指定所有可选的有效数据。列的定义是元数据的一部分，也就是表结构定义的一部分。

比如说，你可以对某一列定义一个检查约束项。这个约束不允许往列中插入或者更新任何会导致约束失败的值。

> 31-Flavors/anti/create-table-check.sql

```
CREATE TABLE Bugs (
  -- other columns
  status VARCHAR(20) CHECK (status IN ('NEW', 'IN PROGRESS',
);
```

MySQL支持使用ENUM关键字来约束一列的取值范围，但这并不是一个标准类型。

> 31-Flavors/anti/create-table-enum.sql

```
CREATE TABLE Bugs (
  -- other columns
  status ENUM('NEW', 'IN PROGRESS', 'FIXED'),
);
```

在MySQL的实现中，即使你使用字符串表示ENUM里的值，但实际存储在列中的数据是这些值在定义时的序数。因此，这列的数据是字节对齐的，当你进行一次排序查询时，结果是按照实际存储的序号进行排序的，而非对应字符串的字母序。这可能并不是你所希望的。

其他的解决方案包含了域以及用户自定义类型（UDT）。你可以使用这些方法来约束某一列只能接受一个特定集合中的数据，并且能很方便地将这个约束应用到整个域上。但这些特性并没有得到大多数关系数据库系统的支持。

最后一个办法，你还可以编写一个触发器，当修改指定列的内容时触发，将被修改的值和允许输入的值进行匹配，如果不符合则产生一个错误中断操作。

所有的这些解决方案都有一定的缺陷。下面就来一一描述这些问题。

11.2.1　中间的是哪个

假设你正在为Bug跟踪服务开发一个用户界面程序，它允许用户编辑Bug报告。为了让界面能够引导用户选择一个合适的status值，你选择使用一个包含所有可选值的下拉菜单。你要如何查询数据库来获取当前可以输入到status列中值的枚举列表呢？

你的第一反应可能是查询当前列中所有正在被使用的值，使用如下这个简单的查询：

31-Flavors/anti/distinct.sql

```
SELECT DISTINCT status FROM Bugs;
```

然而，如果所有的Bug都是新建的，上面这个查询只会返回NEW。如果你使用这样的结果集来填充用户界面中的控件，这就变成了一个先有鸡还是先有蛋的问题：你无法将一个Bug的状态改变为当前使用状态以外的值。

要获得所有允许输入的status候选值，你需要查询这一列的元数据。大多数SQL数据库支持使用系统视图来完成这种查询需求，但是使用起来是很复杂的。比如，如果你使用MySQL的ENUM类型，可以使用如下的查询语句来查询INFORMATION_SCHEMA系统视图：

31-Flavors/anti/information-schema.sql

```
SELECT column_type
FROM information_schema.columns
WHERE table_schema = 'bugtracker_schema'
  AND table_name = 'bugs'
  AND column_name = 'status';
```

你无法简单地从INFORMATION_SCHEMA的结果集中获取单独的枚举值，而是得到了一个包含Check约束或者ENUM类型声明的字符串。比如，在MySQL中，上述查询就会返回一个类型为LONGTEXT、内容为ENUM('NEW', 'IN PROGRESS', 'FIXED')的结果，结果中包含了括号、逗号及单引号。你必须额外编写一段程序代码来解析这个字符串，将每个引号对中的数据独立抽取出来，才能在控件中使用。

对于获取Check约束、域或者UDT信息的查询来说，过程会更加复杂。大多数开发人员非常勇敢地在程序中手动维护这样一个列表。当程序数据和数据库的元数据不同步时，程序很容易就崩溃了。

11.2.2　添加新口味

最常见的改变就是添加或删除一个候选值。没有什么语法支持从ENUM或者Check约束中添加或删除一个值，你只能使用一个新的集合重新定义这一列。如下是一个更新MySQLENUM的例子，目的是将DUPLICATE添加到status的候选值列表中：

31-Flavors/anti/add-enum-value.sql

```
ALTER TABLE Bugs MODIFY COLUMN status
  ENUM('NEW', 'IN PROGRESS', 'FIXED', 'DUPLICATE');
```

你先要知道之前的定义允许NEW、IN PROGRESS和FIXED，这样问题又绕回到了之前如何获取这些值上了。

一些数据库要求只有表为空表时才能改变某一列的定义。你可能需要先转存这张表中的数据，清空这张表，重新定义它，然后再将数据重新导入，这个过程会使得这张表处于无法访问的状态。这种工作非常常见，以致它已经有了个名字：ETL，表示“抽取、转换和加载”。有一些数据库支持使用ALTER TABLE指令重新构建一张使用中的表，

但这个操作依旧是非常复杂的，并且其开销也很巨大。

作为一个策略问题，修改元数据——意味着修改表或列的定义——应该是极少的，并且需要大量的测试以保证质量。如果你需要修改元数据来对一个ENUM定义进行添加或删除操作，就不得不投入大量的测试时间或者让很多工程师关注这个修改所带来的影响。否则，这样的修改就会导致额外的风险，让你的程序变得不稳定。

11.2.3　老的口味永不消失

如果你打算废弃一个选项，你可能会为老数据而烦恼。比如，将质量控制流程中的FIXED状态拆分成CODE COMPLETE和VERIFIED两个状态：

> 31-Flavors/anti/remove-enum-value.sql

```
ALTER TABLE Bugs MODIFY COLUMN status
  ENUM('NEW', 'IN PROGRESS', 'CODE COMPLETE', 'VERIFIED');
```

如果移除FIXED，你要如何处理已经是FIXED状态的数据？将所有的FIXED状态修改为VERIFIED？还是将其设为空或者默认值？

你可能不得不保留这个老数据已使用的废弃选项。但又要如何在用户界面上区分废弃的和可用的Bug状态候选值呢？

11.2.4　可移植性低下

Check约束、域和UDT在各种数据库中的支持形式并不统一。ENUM是MySQL独有的特性。每一个数据库对于列的候选值列表长度的定义都不尽相同。触发器的语法也区别很大。这些差异使得你在需要支持多种数据库时很难选择一个合适的解决方案。

11.3　如何识别反模式

使用ENUM或者Check约束时遇到的问题可能是候选值集合并不固定。如果你正考虑使用ENUM，首先问一下自己，候选值的集合是否需要改变或者是否可能改变。如果答案为“是”，那使用ENUM就不见得是好主意。

- “我们不得不将数据库下线，才能在程序菜单中加入一个新的选项。如果一切顺利，整个过程将不超过3小时。”

  这说明候选值的集合是直接写入列的定义中的。然而，完成这样的升级操作理应不停止服务。

- “这个status列可以填入这些候选值中的一个。我们不应该改变这个候选值列表。”

  “不应该”是一个很模棱两可的说法，它和“不能”是完全不同的意思。

- “程序代码中关于业务规则的选项列表和数据库中的值又不同步了！”

  这就是在两个地方维护同一套数据的风险。

11.4　合理使用反模式

就像我们讨论的那样，ENUM在候选值几乎不变的情况下所造成的问题很少。通过查询获取元数据依旧是很麻烦的，但是你可以在程序代码中维护一份列表而不用担心不同步的问题。

ENUM在存储没有业务逻辑且不需要改变的候选值时是非常方便的。比如存储一对二选一且相互对立的值：LEFT/RIGHT、ACTIVE/INACTIVE、ON/OFF、INTERNAL/EXTERNAL等。

Check约束可以在更多的场景下使用，不仅仅是实现一个类ENUM的机制，比如用来检查一个时间区间中start永远小于end。

11.5　解决方案：在数据中指定值

有一个更好的解决方案来约束一列中的可选值：创建一张检查表，每一行包含一个允许在Bugs.status列中出现的候选值；然后定义一个外键约束，让Bugs.status引用这个新表。

31-Flavors/soln/create-lookup-table.sql

```
CREATE TABLE BugStatus (
  status  VARCHAR(20) PRIMARY KEY
```

```
);

INSERT INTO BugStatus (status) VALUES ('NEW'), ('IN PROGRESS'

CREATE TABLE Bugs (
  -- other columns
  status  VARCHAR(20),
  FOREIGN KEY (status) REFERENCES BugStatus(status)
    ON UPDATE CASCADE
);
```

当你插入或更新Bugs表中的一条记录时，必须使用存在于BugStatus表中的一个status值。这样做类似于ENUM和Check约束一样，能够确保status值的有效性；同时，这样的方案还在其他地方体现出了它的灵活性。

11.5.1 查询候选值集合

候选值集合现在是存储在数据表中，而不是像ENUM那样存储在元数据中。你可以使用SELECT对这张检查表进行查询来获取相关数据，和查询别的表没什么区别。这使得获取候选值集合并作为数据集在用户界面中展示变得非常简单。你甚至可以对用户可选值进行排序操作。

31-Flavors/soln/query-canonical-values.sql

```
SELECT status FROM BugStatus ORDER by status;
```

11.5.2 更新检查表中的值

当你使用检查表时，可以使用原始的INSERT语句向其中加入一个值。你可以不中断对表的访问就完成这样的改变。你也不需要重新定义任何列，不需要安排下线时间，或者执行一次ETL操作。同样，你也不需要在执行添加或删除操作前知道当前检查表中有哪些值。

31-Flavors/soln/insert-value.sql

```
INSERT INTO BugStatus (status) VALUES ('DUPLICATE');
```

如果定义外键时使用了ON UPDATE CASCADE选项，重命名一个值也会变得非常方便。

31-Flavors/soln/update-value.sql

```
UPDATE BugStatus SET status = 'INVALID' WHERE status = 'BOGUS
```

11.5.3 支持废弃数据

如果检查表中的一个值被Bugs表中的数据引用了，那就不能删除它了。status列上的外键确保了引用完整性，因此，status列引用的值必须存在于检查表中。

然而，你可以在检查表中增加另一个属性列来标记一些废弃数据。这样做允许你保留Bugs.status列中的历史数据，同时又能够区分哪些值是能够出现在用户界面上的。

31-Flavors/soln/inactive.sql

```
ALTER TABLE BugStatus ADD COLUMN active
  ENUM('INACTIVE', 'ACTIVE') NOT NULL DEFAULT 'ACTIVE';
```

使用UPDATE代替DELETE来废弃一个值：

31-Flavors/soln/update-inactive.sql

```
UPDATE BugStatus SET active = 'INACTIVE' WHERE status = 'DUPL
```

当要获取在界面上展示的候选值列表时，在查询条件中增加一个status为ACTIVE的约束：

31-Flavors/soln/select-active.sql

```
SELECT status FROM BugStatus WHERE active = 'ACTIVE';
```

这样的解决方案相比于ENUM或者Check约束来说更加灵活，因为前两者无法为每个值提供额外属性。

11.5.4　良好的可移植性

不同于ENUM类型、Check约束，或者域及UDT，检查表的解决方案只依赖于最基本的SQL特性：使用外键确保引用完整性。这使得该解决方案的兼容性得到了保证。

由于在每一行中存储一个候选值，就使得检查表在理论上可以支持无限多个候选值。

> 在验证固定集合的候选值时使用元数据。
>
> 在验证可变集合的候选值时使用数据。

第12章　幽灵文件

> 当一个理论看上去像是唯一可能的理论时，那意味着你既不理解这个理论，也不理解它所要解决的问题。
>
> 卡尔·波普尔

你的数据库服务器在大灾难中没有幸存下来，在清理时发现硬盘机架整个倾倒了，并且摔坏了。幸运的是，没有人因此而受伤，但是大量的硬盘因此而损坏，甚至存放机架的楼层在机架倾倒时被砸穿了。所幸，IT部门的灾备做得比较好：他们每天都为每个重要的系统做了备份，并且快速地在新的服务器上部署了新的服务，恢复了数据库。

冒烟测试进行没多久后发现了一个问题：你的程序将图像与很多数据库字段进行了关联，但是所有这些图片都不见了！你立刻打给了IT技术部门。

“我们恢复了数据库并且验证了这是和上次备份一样的完整副本，”这个技术人员说：“图片是存在哪里的？”

你现在想起来了，在这个应用中，图片是存在数据库之外的，普通文件都是存在文件系统中的。数据库里只存了图片的路径，通过程序去打开对应路径的图片。“图片是以文件形式存储的。它们是在/var目录下，和数据库一起。”

这个技术人员摇了摇头。“除非你做过特殊说明，否则我们不备份/var下的内容。当然我们会备份数据库的文件，但/var目录经常是存放日志、缓存或其他临时文件的地方。默认情况下，这些数据都是不备份的。”

你心痛啊。那个目录下存了超过11 000张在产品分类数据库中使用的图片。大多数可能在其他地方还有备份，但要把它们都放到一起，重新编排，并且为网络搜索重做缩略图要花好几个星期啊！

12.1　目标：存储图片或其他多媒体大文件

如今，图片等多媒体文件已经广泛使用在很多程序中。有时，多媒体文件和数据库中的一些实体关联。比如，你可能会允许一个用户在提交评论时显示一个头像。在我们的Bug数据库中，Bug通常需要附带一个截屏来展示具体情况。

本章的目标就是要存储这些图片并且将其和数据库实体（诸如用户账户或者Bug）关联起来。当查询这些实体时，我们需要确保同时能获取与其关联的图片。

12.2　反模式：假设你必须使用文件系统

理论上来说，图片是一张表中的一个字段，在Accounts表中可能会有一个portrait_image列。

Phantom-Files/anti/create-accounts.sql

```
CREATE TABLE Accounts (
  account_id       SERIAL PRIMARY KEY
  account_name     VARCHAR(20),
  portrait_image   BLOB
);
```

同样地，你可以在从属表中存储多张同类型的图片。比如，每个Bug都可以有多张屏幕截图。

Phantom-Files/anti/create-screenshots.sql

```
CREATE TABLE Screenshots (
```

```
  bug_id              BIGINT UNSIGNED NOT NULL,
  image_id            SERIAL NOT NULL,
  screenshot_image    BLOB,
  caption             VARCHAR(100),
  PRIMARY KEY         (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

这些都不难理解，但真正的重点问题是选择什么样的数据类型来存储图片？原始图片文件可以以二进制格式存储在BLOB类型中，就像之前我们存储超长字段那样。然而，很多人选择将图片存储在文件系统中，然后在数据库里用VARCHAR类型来记录对应的路径。

Phantom-Files/anti/create-screenshots-path.sql

```
CREATE TABLE Screenshots (
  bug_id              BIGINT UNSIGNED NOT NULL,
  image_id            BIGINT UNSIGNED NOT NULL,
  screenshot_path     VARCHAR(100),
  caption             VARCHAR(100),
  PRIMARY KEY         (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

开发人员激烈地争论着这个问题。两种方案都有很好的立足点，但是对于程序员来说通常只有一种选择，就是我们应该将文件存在数据库之外。可能我的观点有点不合时宜，但我依旧要在接下来的几节中指明这样的设计所面临的很现实的风险。

12.2.1 文件不支持DELETE

第一个问题就是垃圾回收。如果图片在数据库之外，并且你想要删除包含这个路径的记录，没有什么办法能自动地将路径对应的文件移除。

Phantom-Files/anti/delete.sql

```
DELETE FROM Screenshots WHERE bug_id = 1234 and image_id = 1;
```

除非你的应用程序设计成在删除记录的同时删除这些“无人领养”的图

片，不然这些图片就会堆积在那里。

12.2.2 文件不支持事务隔离

通常，当更新或删除数据时，在使用COMMIT指令完成事务操作之前，所有的改变都对其他的客户端不可见。

然而，任何对数据库之外的文件的操作并非如此。如果你删除了一个文件，对于其他的客户端来说就立刻无法访问该图片了。并且如果你改变了文件的内容，其他的客户端可以立刻看到这些变更，而不是看到在事务未提交之前的文件状态。

Phantom-Files/anti/transaction.php

```
<?php

$stmt = $pdo->query("DELETE FROM Screenshots
                     WHERE bug_id = 1234 AND image_id =1");

unlink('images/screenshot1234-1.jpg');
// Other clients still see the row in the database,
// but not the image file.

$pdo->commit();
```

在实际生产过程中，这样的特例可能并不常见。同样，本例的影响也比较小；在Web程序中，丢失图片的情况很少见。但在别的情况下，后果就可能非常严重。

12.2.3 文件不支持回滚操作

出错情况下，或者程序逻辑要求取消变更时，对数据进行回滚操作是很平常的事情。

比如，你最开始执行了一句DELETE语句来删除一条记录并同时移除了对应的截屏文件，然后你回滚了这个操作，被删除的数据回来了，但文件已经没了。

Phantom-Files/anti/rollback.php

```
<?php

$stmt = $pdo->query("DELETE FROM Screenshots
                     WHERE bug_id = 1234 AND image_id =1");

unlink("images/screenshot1234-1.jpg");

$pdo->rollback();
```

数据库中的记录能够恢复，但文件不能。

12.2.4　文件不支持数据库备份工具

大多数数据库产品都提供客户端工具来协助备份使用中的数据库。比如MySQL提供了一个叫做mysqldump的组件，Oracle提供了rman，PostgreSQL提供了pg_dump，SQLite提供了.dump命令等。使用备份工具非常重要，如果同一时刻别的客户端正在进行变更操作，将可能使你的备份包含不完整的变更，造成潜在的引用不完整性，甚至使得整个备份被破坏以至于无法用于恢复数据库。

但是备份工具并不知道如何将通过路径引用的那些文件也包含在备份操作当中。因此在备份一个数据库时，你需要记住执行一个两步操作：使用数据库备份工具，然后使用文件系统备份工具来收集外部图像文件。

即使在备份时包含了外部文件，也很难保证这些文件备份和你执行备份数据库的事务是同步的。程序可能在任何时间对图片文件做变更，也许就在你开始备份数据库之后不久。

12.2.5　文件不支持SQL的访问权限设置

外部文件会绕开通过GRANT和REVOKE SQL语句设定的访问权限。SQL权限管理着对表和列的访问，但它们并不能应用到外部文件。

12.2.6　文件不是SQL数据类型

在screenshot_path字段中存储的路径就是一个字符串。数据库并不会验证这个字符串是一个有效的路径，也不会验证对应的文件是否存在。如果这个文件被重命名、移动或者删除了，数据库并不会自动更新对应的路径。任何将这个字符串作为路径处理的逻辑都依赖于你的程序逻辑。

Phantom-Files/anti/file-get.php

```
<?php

define('DATA_DIRECTORY', '/var/bugtracker/data/');

$stmt = $pdo->query("SELECT image_path FROM Screenshots
                     WHERE bug_id = 1234 AND image_id = 1");
$row = $stmt->fetch();
$image_path = $row[0];

// Read the actual image -- I hope the path is correct!
$image = file_get_contents(DATA_DIRECTORY . $image_path);
```

使用数据库的好处之一在于它能帮助我们保持数据完整性。当你将数据放在外部文件中时，就抛弃了数据库提供的这个好处，并且你不得不写更多的程序代码来执行本该由数据库进行的检查操作。

12.3 如何识别反模式

要发现这个反模式需要一定的调查。如果一个项目有指导软件管理员的文档，或者你有机会与设计项目的程序员（或者就是你自己）面谈，考虑一下如下几个问题的答案。

- 数据备份和恢复的过程是怎样的？怎么对一个备份进行验证？你有没有在一个干净的系统或者在别的系统上对备份恢复的数据进行测试？

- 图片文件堆积在那里，还是当它们孤立的时候就从系统中移除？移除它们的过程是怎样的？这是一个自动的还是手动的过程？

- 系统中的哪些用户有权限查看这些图片？进入权限是怎么限制的？当用户请求看他们无权查看的图片时会发生什么？

- 我能撤销对图片的变更吗？如果能，是应用程序来负责恢复图片之前的状态吗？

典型的使用反模式的项目通常没有考虑以上的几个或者全部的问题。并不是每个程序都需要有针对图片的很强的事务管理或者SQL访问控制。可能在备份过程中将数据库下线也是一个很好的方案。如果以上

这些问题的回答不明确或者并没有立刻给出回答，那可以假设这个项目对于外部文件使用的并没有经过精心设计。

12.4 合理使用反模式

如下是一些将图片或者大文件存储在数据库之外的好理由。

- 这个数据库在没有图片的时候能精益很多，因为图片相比于简单的数据类型（比如整形和字符串）来说更大。

- 当不包含图片时备份数据库会更快并且备份的文件更小。你必须额外执行一次文件备份，但这比备份一个大型数据库要更容易管理。

- 如果图片是存储在数据库之外的文件系统里，对图片的预览或者编辑就能够使用更简单直接的处理方式。比如，如果你需要执行一个批处理编辑所有的图片，将其保存在数据库之外就是一个特别好的选择。

如果这些将图片存在文件系统中的好处是重要的，并且之前几节所描述的那些事项并不会破坏你的系统，那就可以肯定将图片存在数据库之外对于你的项目来说是正确的决定。

一些数据库产品提供了一些特殊的SQL数据类型，为支持对外部文件引用提供或多或少的透明性。Oracel称这种类型为`BFILE`，SQL Server 2008称之为`FILESTREAM`。

不要排除任何设计

> 我在1992年的时候为一个外包工程设计了一个将图片存储在数据库之外的程序。我的雇主承接了一个技术会议的注册程序。在会议即将开始之前，我们用一个照相机对每个与会人士拍照，并将他们的照片加到他们的注册信息中，同时也打印在他们的通行证上。
>
> 我的程序非常简单。每个图片都只能被一个客户端插入或者更新（如果这个人在拍照时眨眼了，或者不喜欢他们的照片，我们会在注册时就更换它）。没有复杂的事务处理的需求，也没有多客户端的并发访问或者回滚需求。我们甚至没有使用SQL的权限控制。预览图片的逻辑简单到根本不需要将其从数据库中提取出来。

> 我开发这个项目的时候，数据库及客户端技术还不像现在这么发达。于是我们有很充分的理由在这种情况下将图片直接存在文件系统目录中，并且使用应用层代码来维护。

你需要规划你的程序如何使用这些文件，并且了解本章所描述的反模式是否会影响到你的系统。做一个明智的决定，而不仅仅听那些将图片存在数据库之外的程序员的泛泛之谈。

12.5 解决方案：在需要时使用BLOB类型

如果在12.2节中所描述的任何问题适用于你的项目，你应该要考虑将图片从数据库之外转移到数据库之内。所有的数据库产品都支持BLOB类型，支持你存储任何二进制数据。

> Phantom-Files/soln/create-screenshots.sql

```
CREATE TABLE Screenshots (
  bug_id            BIGINT UNSIGNED NOT NULL,
  image_id          BIGINT UNSIGNED NOT NULL,
  screenshot_image  BLOB,
  caption           VARCHAR(100),
  PRIMARY KEY       (bug_id, image_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

如果你将图片存在一个BLOB类型的列中，所有的问题都将得到解决。

- 图片数据存储在数据库中。不需要额外的步骤加载它，也就没有“文件路径不正确”这样的风险。
- 删除一条记录同时也自动地删除了图片。
- 直到你提交了事务，否则对图片的改变是对其他客户端不可见的。
- 回滚事务会恢复图片之前的状态。
- 更新记录的时候会加锁，因此不会有别的客户端并发更新图片。

- 数据库备份会包含所有的图片。
- SQL权限控制对图片也有效。

BLOB类型的最大值依据不同的数据库产品而不同，但对于存储大部分的图片文件来说都是足够的。所有的数据库都支持BLOB或者类似的类型。比如，MySQL提供了一个叫做MEDIUMBLOB的类型，支持最大16MB的数据，对于绝大部分图片来说都足够了。Oracle提供了一个LONG RAW或者BLOB的类型，最大支持2GB或者4GB的长度。类似的类型在其他数据库产品中也能找到。

图片文件最开始都是以文件形式存储的，因此你需要一些途径将它们载入BLOB列。一些数据库产品提供了加载外部文件的函数。比如，MySQL有个函数叫做LOAD_FILE()，你可以用它来读取一个文件，然后将内容存到BLOB列中。

Phantom-Files/soln/load-file.sql

```
UPDATE Screenshots
SET screenshot_image = LOAD_FILE('images/screenshot1234-1.jpg
WHERE bug_id = 1234 AND image_id = 1;
```

同样地，你也可以将BLOB列的内容存储到一个文件中去。比如，MySQL有一个可选的SELECT子句能将一个查询完整地不加修改地存储到文件中去。

Phantom-Files/soln/dumpfile.sql

```
SELECT screenshot_image
INTO DUMPFILE 'images/screenshot1234-1.jpg'
FROM Screenshots
WHERE bug_id = 1234 AND image_id =1;
```

你也能够直接从BLOB字段中提取图片并且输出。在Web程序中，你可以将二进制数据以图片输出，但你需要将内容类型设置为合适的值。

Phantom-Files/soln/binary-content.php



```
<?php

header('Content-type: image/jpg');

$stmt = $pdo->query("SELECT screenshot_image FROM Screenshots
                     WHERE bug_id = 1234 AND image_id = 1");
$row = $stmt->fetch();

print $row[0];
```

> 存储在数据库之外的数据不由数据库管理。

第13章　乱用索引

> 无论何时，在机器的帮助下找到结果，另一个问题就冒出来了——通过何种计算过程，能让机器最快地求出结果？
>
> 查尔斯·巴贝奇，《哲学家的生命历程》（1864）

“嗨！你有没有时间？我需要你的帮助。”带有俄克拉何马口音的人在电话那头对着你喊，他的声音也顺着数据中心的通风管道传过来。这是你们公司的数据库管理员的头儿。

“当然。”你有点心虚地回答道。他想干嘛？

“是这样，有一个你们的数据库服务在运行，它占了太多资源。”这个DBA继续说道，“我进去看了一下，然后发现了问题。在有些表上完全没有使用索引，而在其他表上索引又多得用不了。

“我们必须解决这个问题，或者给你们一台独立服务器，因为这台机器上的其他服务都无法正常使用了！”

“我很抱歉。事实上，我对数据库了解得不多。”你小心地回答，尽量让这个DBA冷静下来，“我们已经尽了最大的努力来猜测哪些是可以优化的，但显然这些工作只有你们这些专家才能做得好。有没有什么你能做的调整？”

“孩子，我做了所有能做的调整，这就是为什么到目前为止它还没有挂掉，”这个DBA回答道，“留给我的唯一选择是限制你们的程序对数

据库的访问量，但我认为你们并不想要这样。我们必须停止猜测，首先我们要弄明白你们的程序到底需要数据库做什么。”

你觉得头都大了，小心谨慎地问道：“你有什么想法吗？坦白地说，我们组里并没有数据库方面的专家。”

“这不成问题。”这个DBA笑道，“你们确实对你们的程序了如指掌，对吧？而这部分是关键内容，不过我可帮不上忙。我会派一个手下到你们那去帮忙安装一些正确的工具，然后我们将找到并消除你们的瓶颈。你们只需要一点点的指导。你很快就会明白的。”

13.1 目标：优化性能

性能是我从那些数据库开发人员那里听到的一个最普遍的问题。只要看一下任何技术会议的议题你即可明白：到处都充斥着工具和技术来让数据库再多工作一点。当我要做的讲座涉及如何规划数据库或写出更为可靠、安全、正确的SQL方面内容时，有可能他们唯一的疑问在于：“好吧，但是这些对性能有什么影响？”而我对此并不感到吃惊。

改善性能最好的技术就是在数据库中合理地使用索引。索引也是数据结构，它能使数据库将指定列中某个值快速定位在相应的行。索引提供了一种简单而高效的途径让数据库能够快速地找到需要的值，而不是野蛮地进行一次自上而下的全表遍历。

软件开发人员通常并不理解如何或何时使用索引。关于何时使用索引这个问题，数据库相关文档和书籍也很少或根本没有清晰的说明，开发人员通常只能猜测如何有效地使用索引。

13.2 反模式：无规划地使用索引

当我们通过猜测来选择索引时，不可避免地会犯一些错误。对何时使用索引的误解可能会导致如下三种类型的错误：

- 不使用索引或索引不足；
- 使用了太多的索引或者使用了一些无效索引；
- 执行一些让索引无能为力的查询。

13.2.1 无索引

我们通常都知道，数据库在保持索引同步的时候会有额外的开销。我们每次使用`INSERT`、`UPDATE`，或者`DELETE`时，数据库就不得不更新索引的数据结构来使得所记录的表数据是一致的，然后我们使用这个索引进行查询时就能准确地找到所需要的记录。

我们已经习惯性地将额外开销认为是浪费。因此当我们知道数据库在保持索引同步的时候会造成额外的开销，就想要消除它。一些开发人员于是得出结论，终极的解决方案就是不使用索引。这个建议很常见，但它忽略了一个事实，那就是索引能够通过带给你更多的好处来抵销它的额外开销。

> 索引不是标准
>
> 你知道ANSI SQL的标准中没有任何和索引相关的描述吗？数据存储方面的实现和优化在SQL语言中并没有明确的说明，因此每个数据库产品在实现索引的方式上都有很高的自由度。
>
> 大多数数据库产品都使用相似的`CREATE INDEX`语句，但每个产品都有一定的灵活度来修改并加入一些它们自己的技术。索引没有标准的兼容模式。同样地，也没有索引维护、自动化查询优化、查询计划报表的标准，或者类似于`EXPLAIN`这样的指令。
>
> 要尽可能地了解索引，你只能仔细阅读你所使用的数据库产品的文档。具体的语法和特性对于不同的产品来说有很大的区别，但基本的逻辑和理论都是通用的。

不是所有的额外开销都是浪费。你的公司所雇佣的管理层、法律顾问、会计，那些与办公设施有关的开销，甚至那些对公司收入没有直接帮助的开销都是浪费吗？不，因为这些人和物都从不同的方面为公司做了重要的贡献。

在传统的软件中，每一次更新都会执行上百次表查询操作。每次当你执行一个使用索引的查询时，你就抵消了一部分由于维护索引而造成的额外开销。

索引也能帮助`UPDATE`或者`DELETE`语句快速地找到对应的记录。比如，`bug_id`这个主键上的索引能有效提升下面这条语句的效率：

> Index-Shotgun/anti/update.sql



```
UPDATE Bugs SET status = 'FIXED' WHERE bug_id = 1234;
```

一条在非索引列上执行的语句会导致数据库执行全表遍历来查找匹配记录。

Index-Shotgun/anti/update-unindexed.sql

```
UPDATE Bugs SET status = 'OBSOLETE' WHERE date_reported < '20
```

13.2.2　索引过多

你只能在使用索引进行查询时才能受益。对于那些你不使用的索引，你无法获得任何好处。这里有一些例子：

Index-Shotgun/anti/create-table.sql

```
    CREATE TABLE Bugs (
      bug_id         SERIAL PRIMARY KEY,
      date_reported  DATE NOT NULL,
      summary        VARCHAR(80) NOT NULL,
      status         VARCHAR(10) NOT NULL,
      hours          NUMERIC(9,2),
①     INDEX (bug_id),
②     INDEX (summary),
③     INDEX (hours),
④     INDEX (bug_id, date_reported, status)
    );
```

在之前的例子里，有这么几个无用的索引：

① `bug_id`：大多数数据库都会自动地为主键建立索引，因此额外再定义一个索引就是一个冗余操作。这个额外定义的索引并无任何好处，它只会成为额外的开销。关于何时自动创建索引，每个数据库产品都有自己的规则。你需要仔细地阅读你所使用的数据库的说明文档。

② `summary`：对于长字符串，比如`VARCHAR(80)`这种类型的索引要比更为紧凑数据类型的索引大很多。同样地，你也不太可能对`summary`列进行全匹配查找。

③ `hours`：这是另一个你不太可能按照特定值进行搜索的列。

④ `bug_id`，`date_reported`，`status`：使用组合索引是一个很好的选择，但大多数人创建的组合索引通常都是冗余索引或者很少使用。同样地，组合索引中的列的顺序也很重要：你需要在查询条件、联合条件或者排序规则上使用定义索引时的从左到右的顺序。

对冲风险

> 比尔·科斯比说了一个他在拉斯维加斯的故事。他在赌场输了钱之后觉得很有挫败感，于是决定在走之前至少要赢一点东西。因此他买了200美元的25分硬币筹码，然后走到了轮盘赌桌前，在每个方格内，不管红的或是黑的，都放了些筹码，把整张赌桌都覆盖了。然后庄家转动了那个球……那个球掉在了地上。

有些人为每一列——以及每个组合列——都创建索引，因为他们不知道哪个索引对查询有帮助。如果你对数据库中的每张表每个列都做索引，就造成了很多无法确定收益的额外开销。

13.2.3　索引也无能为力

接下来常犯的一个错误就是进行一个无法使用索引的查询。开发人员创建了越来越多的索引，尝试着去发现一个神奇的组合方式或者索引选项来加速他们的查询。

我们可以想象数据库的索引使用了类似于电话号码簿的结构。如果我让你去电话号码簿里查一下有哪些人姓Charles，这会是一个很简单的任务。所有同姓的人都列在了一起，因为这就是电话号码簿排序的方式。

然而，如果我让你去查一下谁的名字叫做Charles，电话簿的名字排序方式就帮不上忙了。任何人都可以叫Charles，而不管他姓什么，因此你必须一行行地在号码簿中查找。

电话号码簿是按照先姓后名的方式对联系人进行排序的，就像一个按照`last_name`、`first_name`顺序创建的联合索引一样。这种索引不能帮你按照名来进行快速查找。

> Index-Shotgun/anti/create-index.sql



```
CREATE INDEX TelephoneBook ON Accounts(last_name, first_name)
```

下面有一些无法使用索引进行查询的例子。

- `SELECT * FROM Accounts ORDER BY first_name, last_name;`

  这个查询就是之前电话号码簿的情况。如果你创建了一种先last_name再first_name顺序的联合索引（就如同电话号码簿），它是不会帮你先按照first_name进行排序。

- `SELECT * FROM Bugs WHERE MONTH(date_reported) = 4;`

  即使你为date_reported列创建了一个索引，这个索引的排序规则也无法帮你按照月份查询。这个索引是按照完整的日期进行排序的，从年份开始。但每一年都有第四个月，因此，月份为4的数据分散在整张表中。

  一些数据库支持表达式索引，或者针对衍生列进行和普通列一样的索引。但你必须在使用前明确地定义索引的行为，并且这些索引只能在你定义的表达式查询上节省时间。

- `SELECT * FROM Bugs WHERE last_name = '_Charles_' OR first_name = '_Charles_';`

  我们又回到之前那个问题上来了，包括指定名字的行分散在整张表中，无法和我们定义的索引顺序匹配。上面的这个查询和下面的这个查询的结果是一样的：

  ```
  SELECT * FROM Bugs WHERE last_name = 'Charles'
  UNION
  SELECT * FROM Bugs WHERE first_name = 'Charles';
  ```

  该例中的索引能帮助我们快速地按姓查找，但对于按名查找则无能为力。

- `SELECT * FROM Bugs WHERE description LIKE '_%crash%_';`

  由于这个查询断言的匹配子串可能出现在该字段的任何部分，因此即使经过排序的索引结构也帮不上任何忙。

13.3 如何识别反模式

如下几点是使用了“乱用索引”反模式的特征。

- “这是我的查询语句，我要怎样才能让它更快？”

  这可能是最常见的一个关于SQL的问题了，但它缺少了表结构、索引、数据集和性能及优化尺度的细节。没有这些上下文，任何的解答都是臆测。

- “我在每个字段上都定义了索引，为什么它没有变得更快？”

  这是“乱用索引”反模式的经典案例。你尝试了所有的索引，但你是在摸黑打鸟啊！

- “我听说索引会使数据库变慢，所以我不会使用它。”

  就像很多开发人员那样，你在找一个提升性能的万全之策。根本没有这样的好事啊！

13.4 合理使用反模式

如果你需要设计一个普通用途的数据库，不了解哪些查询是需要重点优化的，你就不能确定哪些索引是最好的。你可能需要做一些有根据的猜测。你有可能会漏掉一些能有所帮助的索引，也有可能会创建了一些没用的索引。但你必须尽量去尝试。

> 低分离率索引
>
> 分离率是衡量数据库索引的一个指标。它是一张表中，所有不重复的值的数量和总记录条数之比：
>
> ```
> SELECT COUNT(DISTINCT status) /
>     COUNT(status) AS selectivity FROM Bugs;
> ```
>
> 分离率的值越低，索引的效率就越低。为什么？我们可以想象下面这样的情况。
>
> 这本书有一个关于不同数据类型的索引：索引中的每一条记录都给出这个单词出现的页的列表。如果一个单词在这本书中频繁地

> 出现，就可能会有很多页码。要找到所需要找到的部分，你就不得不翻到这个列表中的每一页去查看。
>
> 索引本身并不会觉得存储那些出现频率很高的单词有什么负担。但如果你需要频繁地在索引和页面间来回查找所需要的内容，你可能就跟从头到尾读了这本书没什么区别。
>
> 数据库的索引机制也是类似的，如果一个给定的值出现在这张表的很多条记录中，查询索引比简单地扫描一遍整张表更麻烦。事实上，使用这个索引的开销可能比不使用它还大。
>
> 你需要时刻关注你的数据库中索引的分离率，并且抛弃那些低效的索引。

13.5　解决方案：MENTOR你的索引

“乱用索引”这个反模式是关于随意创建或抛弃索引的，因此，让我们来分析一下一个数据库，并且找一些好的理由来使用或者丢弃索引。

你可以使用好记的MENTOR方法来分析数据库索引的使用：测量（Measure），解释（Explain），挑选（Nominate），测试（Test），优化（Optimize）和重建（Rebuild）。

> 数据库不一定是瓶颈
>
> 软件开发人员通常的经验是数据库总是程序中最慢的部分，并且是性能问题的根源。然而，事实并非如此。
>
> 举例来说，在我曾经参与过的一个项目中，我的经理让我找出为什么程序跑得这么慢，并且他坚持认为是数据库的错。然而在我用了一个性能测评工具去检测程序代码之后，我发现它用了80%的时间来解析HTML，然后找出表单字段并往里面填入对应的值。这个性能问题和数据库完全无关。
>
> 在对性能问题下结论之前，先用一些分析工具来测试一下。否则你可能就在做一些过度优化。

13.5.1　测量

你不能在没有信息的情况下做出决定。大多数数据库都提供了一些方

法来记录执行SQL查询的时耗，因此可以以此来定位最耗时的查询。

- Oracle和微软的SQL Server都有SQL跟踪功能和工具来生成并分析相应报表。微软称之为SQL Server Profiler，Oracle称之为TKProf。

- MySQL和PostgreSQL会记录耗时超过一个特定值的查询请求。MySQL称之为“慢查询”日志，其配置文件中的`long_query_time`项默认设定为10秒。PostgreSQL有一个类似的配置项叫做`log_min_duration_statement`。

- PostgreSQL还有一个配套的工具叫做pgFouine，它能帮助你对查询日志进行分析,并且定位出那些需要格外注意的查询请求（http://pgfouine.projects.postgresql.org/）。

一旦你知道程序中哪些查询耗时最多，就知道该专注于哪方面的优化才能获得最大的效果。你甚至可能发现除了某一个特定的查询比较慢之外，其他的都很快，那这个查询就是性能的瓶颈。这个查询就是你该优化的对象。

如果一个查询很少被调用，那即使它是单次调用耗时最多的一个，也不见得是最耗时间的查询。其他更简单一点的查询可能被调用得很频繁，比你所预期的还要频繁，因此它们所花费的总时间就会更多。专注于这些能够让你事半功倍的查询优化。

记住在做查询性能测试的时候要禁止所有的查询结果缓存。这些缓存被设计用来绕过查询过程和索引使用，因此，如果不禁止这些缓存，你是得不到准确信息的。

在部署程序之后，你可通过进行profile分析来得到更准确的信息。要在实际用户的使用中收集综合数据来查看到底在哪些地方所花的时间是最长的。你应该时刻监控着这些profile数据，以防止不小心造成了一个新的瓶颈。

记住在分析完了之后要关闭profiler或者降低profiler运行的频率，因为这些工具也会造成额外的开销。

13.5.2　解释

已经找到耗时最多的查询请求了，接下来要做的事情就是找出它之所以会这么慢的原因。每个数据库都使用一种优化工具为每次查询选择

合适的索引。你可以让数据库生成一份它所做分析的报表，我们称之为查询执行计划（QEP）。

每种数据库的请求QEP的语法都不尽相同。

数据库	QEP报表方案
IBM DB2	EXPLAIN, db2expln命令, 或 Visual Explain
Microsoft SQL Server	SET SHOWPLAN_XML, 或 Display Execution Plan
MySQL	EXPLAIN
Oracle	EXPLAIN PLAN
PostgreSQL	EXPLAIN
SQLite	EXPLAIN

QEP的报表中包含什么或者报表的形式是什么，没有统一标准。通常来说，QEP会告诉你在一个查询中需要用到哪些表，优化工具是怎么选择索引的，以及按照什么顺序访问这些表。报表可能也会包含一些统计信息，比如每一阶段查询产生多少行记录等。

让我们来看一个简单的SQL查询并请求QEP报表：

> Index-Shotgun/soln/explain.sql

```
EXPLAIN SELECT Bugs.*
FROM Bugs
JOIN (BugsProducts JOIN Products USING (product_id))
  USING (bug_id)
WHERE summary LIKE '%crash%'
  AND product_name = 'Open RoundFile'
ORDER BY date_reported DESC;
```

图13-1为MySQL的QEP报告，key这一列表明这个查询仅使用了BugsProducts表的主键索引。同时，最后一列的额外信息表明这个查询会在一张临时表中对数据进行排序，没有任何索引优化。

table	type	possible_keys	key	key_len	ref	rows	filtered	Extra
Bugs	ALL	PRIMARY,bug_id	NULL	NULL	NULL	4650	100	Using where; Using temporary; Using filesort
BugsProducts	ref	PRIMARY,product_id	PRIMARY	8	Bugs.bug_id	1	100	Using index
Products	ALL	PRIMARY,product_id	NULL	NULL	NULL	3	100	Using where; Using join buffer

图13-1　**MySQL查询执行计划**

LIKE表达式强制在Bugs表中进行全表遍历，在Products.product_name列上没有索引。我们可以通过在product_name上创建一个新的索引以及改用全文搜索的方案来优化这个查询[1]。

> [1] 参考第17章内容。

QEP的信息是由数据库开发商决定的。在本例中，你应该仔细阅读MySQL手册中“Optimizing Queries with EXPLAIN”一章中的说明来理解对应的报表的含义[2]。

> [2] http://dev.mysql.com/doc/refman/5.1/en/using-explain.html。

13.5.3　挑选

现在已经有了查询优化工具的QEP报表，你应该仔细地查找那些没有使用索引的查询操作。

有些数据库会用一些工具来做这件事，它们会收集查询统计信息并且提出一系列的修改建议，包括创建那些被你漏掉但能起到较好效果的索引。比如：

- IBM DB2 Design Advisor；
- Microsoft SQL Server Database Engine Tuning Advisor；
- MySQL Enterprise Query Analyzer；
- Oracle Automatic SQL Tuning Advisor。

即使没有自动化工具，你也可以学习如何辨识一个索引是否有利于提高搜索效率。你需要仔细阅读你所使用数据库的手册来更好地理解QEP报告。

> 索引覆盖
>
> 如果一个索引包含了我们所需要的所有列，那就不需要再从表中获取数据了。

想象一下，如果电话簿的条目中只包含一个页码，在你找到一个名字之后，你不得不翻到对应的页上才能得到所需要的号码。如果将这个过程整合为单步操作会更合情合理。由于电话簿是排序的，所以查找一个名字是很快的，然后在一个条目中可以再包含一个字段存储电话号码，甚至存储地址。

这就是索引覆盖的作用。你可以定义让一个索引包含额外的列，即使这些列对于这个索引来说并不是必须包含的。

```
CREATE INDEX BugCovering ON Bugs
  (status, bug_id, date_reported, reported_by, summary);
```

如果你查询的列都包含在这个索引的数据结构中，数据库就可以通过只查询索引生成结果。

```
SELECT status, bug_id, date_reported, summary
FROM Bugs WHERE status = 'OPEN';
```

数据库并不需要去查询对应的表。虽然你不能在任何地方都使用索引覆盖，但只要使用它，对性能来说就是一个极大的提升。

13.5.4 测试

这一步非常重要：在创建完索引之后，需要重新跟踪那些查询。需要确认你的改动确实提升了性能，然后就能确定工作完成了。

你可以使用这一步骤来给老板留下好印象，并证明你所作的优化是有效的。你肯定不希望周报上写道："我尝试了每个我想到的办法来解决程序的性能问题，然后我们只能等等看反馈……"相反，你现在有机会这么写周报："我发现可以在一个高活跃度的表上创建一个新的索引，并且我将核心查询性能提高了38%。"

13.5.5 优化

索引是小型的、频繁使用的数据结构，因而很适合将它们常驻在内存中。内存操作的性能是磁盘I/O操作的好几倍。

数据库服务允许你配置缓存所需要的系统内存大小。大多数数据库的默认配置都很小，从而能保证数据库在大部分操作系统上都正常工作。通常情况下我们需要调高这个缓存大小的设置。

需要使用多少内存做为索引缓存？这个问题没有标准答案，因为它取决于你的数据库的规模和服务器的内存大小。

使用索引预载入的方法可能要比通过数据库活动本身将最频繁使用的数据与索引放入缓存更有效一点。比如，在MySQL中，使用`LOAD INDEX INTO CACHE`语句。

13.5.6　重建

索引在平衡的时候其效率最高，当你更新或者删除记录时，索引就逐渐变得不平衡，就如同文件系统随着时间的推移会产生很多磁盘碎片一样。在实际运行中，你可能看不出一个平衡索引和一个有些不平衡的索引的区别。但我们想要最大限度地使用索引，因此要定期对索引进行维护。

就像索引的很多其他特性一样，每个不同的数据库都使用自己特有的术语、语法和功能。

数据库	索引维护命令
IBM DB2	REBUILD INDEX
Microsoft SQL Server	ALTER INDEX ... REORGANIZE, ALTER INDEX ... REBUILD, or DBCC DBREINDEX
MySQL	ANALYZE TABLE or OPTIMIZE TABLE
Oracle	ALTER INDEX ... REBUILD
PostgreSQL	VACUUM or ANALYZE
SQLite	VACUUM

多久需要重建一次索引？你可能听到诸如“每周一次”这样空泛的回答，但事实上对于不同的程序来说并没有统一的答案。具体的时间取决于你对指定的表所做的会引起不平衡操作的频率。同时也取决于这张表有多大以及理想状态的索引是否那么的重要。花上几个小时重建一张很大但很少使用的表的索引，只获得了1%的性能提升，这样做值得吗？所有的判断都需要你来做决定，因为你是最了解这些数据和数据操作需求的人。

很多关于最优化使用索引的知识都是根据不同数据库而论的，因此你需要仔细研究所使用的数据库。你手上所有的资源包括数据库手册、书籍和杂志、博客文章和邮件列表，以及很多自己的经验。最重要的规则是千万别瞎猜索引的使用方法。

> 了解你的数据，了解你的查询请求，然后MENTOR你的索引。

第三部分　查询反模式

本部分内容

- 第14章　对未知的恐惧
- 第15章　模棱两可的分组
- 第16章　随机选择
- 第17章　可怜人的搜索引擎
- 第18章　意大利面条式查询
- 第19章　隐式的列

第14章　对未知的恐惧

> 就像我们所知道的那样，有一些众所周知的事情，我们知道自己已经了解了。我们也知道，有一些未知的事情，我们知道自己还不了解。但还有些没人知道的事情，我们并不知我们还一无所知。
>
> 唐纳德·拉姆斯菲尔德

在我们的范例Bug数据库中，Accounts表有first_name和last_name两列。你可以通过使用字符串链接操作将这两个字段合并成用户的全名。

> Fear-Unknown/intro/full-name.sql

```
SELECT first_name || ' ' || last_name AS full_name FROM Accou
```

假设老板让你修改一下数据库，加上用户的中间名的缩写。（如果两个用户的名和姓是一样的，通过中间名的缩写能够减少误解。）这是个很简单的活。你也可以手动加了一些用户的中间名缩写。

> Fear-Unknown/intro/middle-name.sql



```
ALTER TABLE Accounts ADD COLUMN middle_initial CHAR(2);

UPDATE Accounts SET middle_initial = 'J.' WHERE account_id =
UPDATE Accounts SET middle_initial = 'C.' WHERE account_id =

SELECT first_name || ' ' || middle_initial || ' ' || last_nam
FROM Accounts;
```

突然，程序不显示任何名字了。事实上，看了一下之后，你发现并不都如此。只有那些填了中间名缩写的用户的名字能正常显示，其他人的名字都是空的。

其他人的名字怎么了？你能在老板发现并开始恐慌地猜测你是不是丢了数据之前修复这个错误吗？

14.1 目标：辨别悬空值

不可避免地，你的数据库中总会有一些字段是没有值的。不管是插入一个不完整的行，还是有些列可以合法地拥有一些无效值。SQL支持一个特殊的空值，就是我们所熟知的NULL。

有很多在SQL的表和查询中有效使用空值的途径。

- 你可以在添加一条记录时，使用NULL代替那些还不确定的值，比如一个在职员工的离职时间。
- 一个给定的列如果没有合适的值，可以在对应的行中使用NULL。比如对于一辆完全靠电力驱动的车，它的燃油消耗比就毫无意义。
- 当传入参数无效时，一个函数的返回值也可以是NULL，比如DAY('2009-12-32')。
- 在外联结查询中，NULL被用来当做未匹配的列的占位符。

本章的目的就是弄清楚如何编写那些包含NULL的查询。

14.2 反模式：将NULL作为普通的值，反之亦然

很多开发人员都对SQL中NULL的行为感到茫然无措。SQL将NULL当做一个特殊的值，不同于0、false或者空字符串，这一点和大多数的编程语言都不同。大多数的数据库都遵循这种SQL标准，然而在Oracle和Sybase中，NULL的意义是长度为0的空字符串。NULL这个值也有一些特殊的行为。

14.2.1 在表达式中使用NULL

让人奇怪的是，在一些值为NULL的列上进行计算所得到的结果。比如说，很多开发人员都觉得如下查询请求的结果中10应当作为Bugs的默认值返回，但是当hours列中的值为NULL是，返回的结果还是NULL，并非10。

Fear-Unknown/anti/expression.sql

```
SELECT hours + 10 FROM Bugs;
```

NULL和0是不同的。比未知数大10的数还是未知数。

NULL和空字符串也是不一样的。将一个字符串和标准SQL中的NULL联合起来的结果还是NULL（忽略Oracle和Sybase中的行为）。

NULL和FALSE也是不同的。AND、OR和NOT这三个布尔操作如果涉及NULL，其结果也让很多人感到困惑。

14.2.2 搜索允许为空的列

如下查询仅返回assigned_to为123的行，不包含别的值或者NULL：

Fear-Unknown/anti/search.sql

```
SELECT * FROM Bugs WHERE assigned_to = 123;
```

你可能会觉得如下查询会返回上面那个查询的补集，也就是所有之前查询没有返回的行：

Fear-Unknown/anti/search-not.sql

```

```

```
SELECT * FROM Bugs WHERE NOT (assigned_to = 123);
```

然而，这两个查询都不会返回assigned_to是NULL的记录。任何和NULL的比较都返回“未知”，既不是TRUE也不是FALSE。即使NULL的相反值也是NULL。

下面这个错误是在查询NULL或者非NULL值时常犯的：

Fear-Unknown/anti/equals-null.sql

```
SELECT * FROM Bugs WHERE assigned_to = NULL;

SELECT * FROM Bugs WHERE assigned_to <> NULL;
```

WHERE子句的筛选逻辑是当其条件的返回值为TRUE时选择该记录，但和NULL比较永远得不到TRUE，相应地，返回的是“未知”。无论比较的逻辑是相等还是不等，返回的结果还是“未知”，当然“未知”不等于TRUE。之前的所有查询请求都没办法获得assigned_to为NULL的记录。

14.2.3　在查询参数中使用NULL

在参数化的SQL查询表达式中使用NULL进行查询时，碰到使用NULL作为普通值的列也非常地不方便。

Fear-Unknown/anti/parameter.sql

```
SELECT * FROM Bugs WHERE assigned_to = ?;
```

上述查询在传入一个普通的整形时会返回你所期望的值，但你不能使用NULL作为参数传入。

14.2.4　避免上述问题

处理NULL会使查询变得更加复杂，很多软件开发人员就会选择在数据库中禁止NULL。取而代之的是，他们选择一个普通的值来标记“未知”或者“无效”。

“我们痛恨**NULL**！”

> 杰克，一个软件开发人员，描述了他的客户端开发人员要求他屏蔽数据库中所有的NULL。他们的解释就是简单的“我们痛恨NULL”。NULL的存在会导致他们的程序代码出错。杰克询问我该用哪个值来代替NULL标记悬空值。
>
> 我告诉杰克，NULL的作用正是用来表示悬空值或者无效值。无论他选择别的什么值来标记一个悬空值，都要修改程序代码来对这个值进行特殊处理。
>
> 杰克的客户端开发人员对待NULL的态度是完全错误的。类似地，我可以坦率地说，我也不喜欢写代码来处理将零作为除数的问题，但这并不是不使用零的合理理由。

这样做的实际危害在哪里呢？来看如下的例子，我们将assigned_to这一列声明为NOT NULL：

> Fear-Unknown/anti/special-create-table.sql

```
CREATE TABLE Bugs (
  bug_id              SERIAL PRIMARY KEY,
  -- other columns
  assigned_to         BIGINT UNSIGNED NOT NULL,
  hours               NUMERIC(9,2) NOT NULL,
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

假设我们使用-1表示一个未知的值。

> Fear-Unknown/anti/special-insert.sql

```
INSERT INTO Bugs (assigned_to, hours) VALUES (-1, -1);
```

hours这一列是数字，因此你定义某一个特殊的数字表示“未定义”。这个数字在这列中必须不具有任何含义，所以你选择了一个负数。但是-1这个值在执行SUM或者AVG的时候会导致计算结果错误。因而你必须在计算时使用另一个条件判断来去除这些列，这些额外的工作都是因为你要避免使用NULL所带来的。

> Fear-Unknown/anti/special-select.sql

```
SELECT AVG( hours ) AS average_hours_per_bug FROM Bugs
WHERE hours <> -1;
```

在另一列中，-1可能是一个有意义的值，因此你不得不根据每一列的定义和取值范围逐个选择一个不同的值。你还需要记住或者将这些特殊值写成文档。这都给项目增加了过多的复杂度和不必要的工作。

再来看assigned_to列。这是一个指向Accounts表的外键。当一个Bug被提交进系统但还没有指派给任何人去处理，应该用哪个非NULL的值来表示这个逻辑呢？任何一个非NULL的值都必须指向Accounts表中的一条记录，因此你需要在Accounts表中插入一条占位符似的记录，表示“没有人”或者“未指派”。为了让你能将一个Bug指派给一个人而创建这样一个没有意义的账号看上去非常地可笑。

当你将一列声明为NOT NULL时，也就是说，这列中的每一个值都必须存在且是有意义的。比如说，Bugs.reported_by这一列必须有一个值，因为每个Bug都是由某个人报告的。但一个Bug可能暂时还没有指派给任何人处理。悬空的值应该用NULL来表示。

14.3　如何识别反模式

如果你发现自己或者团队中有人描述了如下事件，可能就是没有正确地处理NULL。

- “我要怎么将assigned_to（或别的列）中没有值的列取出来？”

  你不能对NULL使用等号。我们在本章的后半部分将会看到如何使用IS NULL操作符。

- “我能在数据库中看到用户的名字，但是在程序显示的时候，全名就是空的。”

  问题可能是由于你将字符串和NULL进行了拼接的操作，返回的结果还是NULL。

- “这份报告中这个项目的总工作时间仅包括了很小一部分我们设定了优先级的Bug!”

你的统计时间的合计查询可能包含了一个WHERE子句，其中的表达式会因为priority为null而不会返回TRUE。当你使用“不等于”操作的时候要格外注意异常值。比如，对于priority的值为NULL的记录，priority <> 1这个表达式就不会返回真。

- “现在的情况是，我们不能再使用之前在Bugs表中用来表示未知的那个字符串了，因此我们要开个会讨论该用哪个新的特殊值，再评估一下开发时间以及数据转换的时间。”

  这是使用一个特殊标记值的常见后果，这个特殊值很有可能会变得合法。最终你可能会发现需要使用这个值的字面意义而不是标记意义。

> **NULL也是可关联的吗**
>
> 关于SQL中的NULL也有一些争论。创立关系理论的计算机科学家E. F. Codd认为，使用NULL来标记悬空值是有必要的。然而，C. J. Date展示了标准SQL中定义的NULL的行为在某些特殊情况下和关系逻辑相违背。
>
> 事实是大多数编程语言都没有完美地实现计算机科学的那些理论。无论如何SQL都支持NULL。我们已经发现了一些危害，但你可以学着如何找出这些危害从而更有效地使用NULL。

要辨识出对NULL的错误使用是比较困难的事情。在测试阶段，一些错误可能不会发生，尤其是如果你漏掉了一些边界条件的测试而仅仅构造了一些简单的测试数据。然而，当程序用于实际生产时，所产生的数据可能是你不曾预料到的。如果一个NULL存在于数据中，产生错误就是一个时间问题了。

14.4　合理使用反模式

使用NULL并不是反模式，反模式是将NULL作为一个普通值处理或者使用一个普通的值来 取代NULL的作用。

有一种情况可以将NULL视为普通值，那就是导入或者导出数据的时候。在一个以逗号分割的文本文件中，所有的值都必须是可读的文本。比如，MySQL的mysqlimport工具在从文本文件中导入数据时，使用_\N_代表NULL。

类似地，用户不能直接输入一个NULL。程序的输入端可能会使用一些特殊处理来引导用户输入NULL。比如，微软.NET 2.0及以上版本，为Web界面提供了一个叫做ConvertEmptyStringToNull的方法。参数和绑定字段会自动地将空字符串转换成NULL。

最后，如果需要支持多个不同的悬空值时，NULL也不起作用。比如说，想要将一个Bug分为“从未被指派”和“被指派给一个离开项目组的人”，在这种情况下你不得为每种状态使用不同的值。

14.5　解决方案：将NULL视为特殊值

大多数使用NULL的问题都是源自于对SQL的三值逻辑的误解。对于习惯了传统的true/false逻辑程序员来说，这可是一个不小的挑战。不过只要稍微研究一下，你就能正确地在SQL中处理NULL。

14.5.1　在标量表达式中使用NULL

假设斯坦30岁，而奥利弗的年龄未知。如果我问你到底是斯坦大还是奥利弗大，你只能回答：“我不知道。”如果我问你斯坦是不是和奥利弗一样大，你还是只能回答：“不知道。”如果我问你他们两个加起来有多大，你的答案依旧如此。

假设查理的年龄也是未知。如果我问你奥利弗和查理是不是一样大，你的答案还是“不知 道”。这就是为什么NULL = NULL的结果是NULL。

下表列举了一些程序员期望得到某种结果，但事实却不如人意的情况。

表达式	期望值	实际值	原　　因
NULL = 0	TRUE	NULL	NULL不是0
NULL = 12345	FALSE	NULL	如果未指定值和所给值相等则未知
NULL <> 12345	TRUE	NULL	不相等则未知
NULL + 12345	12345	NULL	NULL不是0
NULL || 'string'	'string'	NULL	NULL不是空字符串
NULL = NULL	TRUE	NULL	未指定值和另一个值相等则未知
NULL <> NULL	FALSE	NULL	如不同则未知

当然，这些例子并不仅仅表示直接使用NULL这个关键字的情况，它

们同样也适用于那些返回值为NULL的表达式。

14.5.2　在布尔表达式中使用NULL

理解NULL在布尔表达式中行为的关键点在于，要明白NULL既不是TRUE也不是FALSE。

下表列举了一些程序员所期望得到某种结果，但事实却不如人意的情况。

表达式	期望值	实际值	原　因
NULL AND TRUE	FALSE	NULL	NULL不是FALSE
NULL AND FALSE	FALSE	FALSE	任何值AND FALSE是伪值
NULL OR FALSE	FALSE	NULL	NULL不是FALSE
NULL OR TRUE	TRUE	TRUE	任何值OR TRUE是真值
NOT (NULL)	TRUE	NULL	NULL不是FALSE

一个NULL值当然不是TRUE，同样也不是FALSE。如果是FALSE，对一个NULL使用NOT操作符，则应该返回TRUE，但事实是NOT(NULL)依旧返回一个NULL。这样的行为让那些想要在布尔表达式中使用NULL的人感到很困惑。

14.5.3　检索NULL值

由于等于或者不等于操作在对NULL进行比较时都不会返回TRUE，你就需要使用别的操作来检索NULL。老的SQL标准定义了一个IS NULL的断言，在一个给定的操作值为NULL时，它将返回TRUE。与之对应的，IS NOT NULL在操作值是NULL时，返回FALSE。

Fear-Unknown/soln/search.sql

```
SELECT * FROM Bugs WHERE assigned_to IS NULL;

SELECT * FROM Bugs WHERE assigned_to IS NOT NULL;
```

在SQL-99标准中，额外定义了一个比较断言IS DISTINCT FROM。这个断言的行为模式类似于原来的<>操作符。不同的是，在处理操作数为NULL时它依旧会返回TRUE或FALSE。

> 错误的方法正确的结果
>
> 在如下的情况中，一个允许为空的列的行为基于巧合而得到了正确的结果。
>
> ```
> SELECT * FROM Bugs WHERE assigned_to <> 'NULL';
> ```
>
> 这里允许为空的assigned_to列正用来和'NULL'这个字符串进行比较（注意引号），而不是和NULL这个关键字比较。
>
> 当assigned_to为NULL时，和字符串'NULL'的比较结果不为TRUE。这一条记录就从查询结果中排除了，而这正是程序员所期望的。
>
> 其余的情况是，一个整型的列和字符串'NULL'进行比较。像'NULL'这样的字符串在大多数数据库中都被视为0，而assigned_to的大多数值都大于0。这样的值就不等于一个字符串，因此在结果集中就有了这条记录。
>
> 因此，由于犯了另一个常见的错误——在NULL关键字上使用了引号——一些程序员可能无意间就得到了他们想要的结果。不幸的是，这样的巧合并不是总发生，比如WHERE assigned_to = 'NULL'。

这个断言能让你在进行比较操作之前不用再编写冗长的表达式判断IS NULL。如下两个查询是等价的：

> Fear-Unknown/soln/is-distinct-from.sql

```
SELECT * FROM Bugs WHERE assigned_to IS NULL OR assigned_to <

SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM 1;
```

你可以和查询参数一起使用这个断言，即使传入值就是一个NULL：

> Fear-Unknown/soln/is-distinct-from-parameter.sql

```
SELECT * FROM Bugs WHERE assigned_to IS DISTINCT FROM ?;
```

每个数据库对IS DISTINCT FROM的支持是不同的。PostgreSQL、IBM DB2和Firebird直接支持它，Oracle和Microsoft SQL Server暂时还不支持。MySQL提供了一个专有的表达式<=>，它的工作逻辑和IS NOT DISTINCT FROM一致。

14.5.4　声明**NOT NULL**的列

如果NULL会破坏程序逻辑或者NULL本身就是毫无意义的，那我推荐你在定义对应的列时加上NOT NULL约束。让数据库来帮你确保约束的实行比自己写代码可靠得多。

比如说，在Bugs表中任意记录的date_reported、reported_by和status这几列的值都应该是非NULL的。类似地，在Comments这张子表中也必须有一个非NULL的bug_id列，指向一个切实存在的Bug。你应该在声明这些列的时候都加上NOT NULL选项。

有些人推荐你为每一列都定义一个DEFAULT值，这样一来当在执行插入操作时即使省略了某一列，也能获得一个非NULL的值。这样的建议也并不是通用的。比如说，Bugs.reported_by应该总是非NULL的。如果要定义默认值，应该用哪个值？对于一个在逻辑上没有默认值的列来说，声明一个NOT NULL约束是很合理、很常见的。

14.5.5　动态默认值

在一些查询请求中，你需要强制让某一列或者某个表达式返回非NULL的值，从而让查询逻辑变得更简单，但又不想将这个值存下来。你所需要的仅仅是在特定的请求时对一个给定的列临时设置一个默认值的方法。为此可以使用COALESCE()函数。这个函数接受一系列的值作为入参，并且返回第一个非NULL的参数。

在本章最开始所描述的拼接用户名字的案例中，可以使用COALESCE()函数来写一个表达式，使用一个空格代替中名缩写，这样即使中名缩写是NULL，返回的结果也始终是一个非NULL的中间名缩写，从而最终的结果不会变成NULL。

Fear-Unknown/soln/coalesce.sql

```
SELECT first_name || COALESCE(' ' || middle_initial || ' ', '
  AS full_name
FROM Accounts;
```

COALESCE()是一个SQL标准函数。一些数据库使用了别的函数来实现这个功能，诸如 NVL()或ISNULL()。

> 使用NULL来表示任意类型的悬空值。

第15章　模棱两可的分组

> 智商在于区分可行与否，理性在于区分明智与否。有时候可行却并不明智。
>
> 马克思·伯恩

假设你的老板想通过Bug数据库来了解哪些项目还处于活跃状态，哪些项目已经停止了。他让你生成的一个每个项目最后一个Bug报告提交日期的报表。你写了个查询过程，在MySQL中查询根据product_id分组的Bug的date_reported最大的值。生成的报表类似于下面这样。

product_name	latest	bug_id
Open RoundFile	2010-06-01	1234
Visual TurboBuilder	2010-02-16	3456
ReConsider	2010-01-01	5678

你的老板是一个注重细节的人，他花了点时间研究这份报告。他注意到，在Open RoundFile这个项目下所列出来的最新的bug_id其实并不是最新的。比较一下全量数据就能看出差异。

product_name	date_reported	bug_id	
Open RoundFile	2009-12-19	1234	这个bug_id...... 与这个日期不匹配
Open RoundFile	2010-06-01	2248	
Visual TurboBuilder	2010-02-16	3456	
Visual TurboBuilder	2010-02-10	4077	
Visual TurboBuilder	2010-02-16	5150	
ReConsider	2010-01-01	5678	
ReConsider	2009-11-09	8063	

你要怎么解释这个问题？为什么它会只影响了一个产品但对于其他的产品来说又是正常的？你要怎么才能得到正确的报告？

15.1　目标：获取每组的最大值

大多数程序员在学习SQL时都会遇到在查询时需要使用`GROUP BY`的情况，比如对分组多行记录使用一些聚合函数，每组获取一条统计记录等。`GROUP BY`的强大之处就在于，它把复杂的报表生成过程简化到只用相对很少的代码。

例如，想要使用单次查询来获取Bug数据库中每一个产品最新的Bug报告，可以这么写：

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

对于上述查询最基本的扩展就是获取最新Bug的ID：

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

然而，这个查询在不同的数据库中，要么是一个语法错误，要么就得到一个不可靠的结果。使用SQL的开发人员对此普遍感到很困惑。

本章的目标就是能执行一个不仅返回每组最大值的查询（或者最小值、平均值），同时也要能返回这个值对应的记录的其他字段。

15.2　反模式：引用非分组列

造成这个反模式的最根本原因很简单，而且它揭示了很多程序员对于SQL中分组查询逻辑的普遍误解。

15.2.1　单值规则

属于每个组的行，它们的`GROUP BY`关键字所指定的那些列中的值都

是一样的。比如下面这个查询，对于每个不同的product_id都会返回一条记录。

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

跟在SELECT之后的选择列表中的每一列，对于每个分组来说都必须返回且仅返回一个值。这就称为单值规则。在GROUP BY子句中出现的列能够保证它们在每一组都只有一个值，无论这一个组匹配多少行。

MAX()表达式也能保证每组都返回单一的值，即返回传入MAX()的参数中最大的那个值。

然而，数据库服务不能确定其他那些在选择列表中的列，它们都是单值的。它不能保证被分在同一组中的行的其他列都共有一个值。

Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

在这个例子中，一个给定的product_id有很多不同的bug_id，因为BugsProducts表将很多Bug和同一个产品关联起来了。在一个根据product_id进行分组的查询中，数据库没办法在查询结果中表示所有的bug_id。

由于对于这些“额外”的列没有办法保证它们满足单值规则，数据库就假设它们违反了单值规则。大多数的数据库在你尝试返回一个不在GROUP BY的参数中的列或者不是由聚合函数返回的值时，抛出一个错误。

MySQL和SQLite在这方面的行为和其他的数据库不同，我们将在15.4节中具体说明。

15.2.2　我想要的查询

程序员中常见的误解是认为SQL能猜测你需要在报告中显示哪个bug_id，因为MAX()是用在另一列而不是GROUP BY的那些列上。大多数人假设如果查询得到了最大值，那么查询返回结果中的其他列就会是对应的这个最大值所在的列中的那些值。

不幸的是，SQL由于下面这几个原因，不会这么做。

- 如果两个Bug的date_reported值相同并且这个值就是这一组中的最大值，哪个bug_id应该放到报告中？
- 如果聚合函数的返回值没有匹配表中的任何一行，又该用哪个bug_id？当使用AVG()、COUNT()和SUM()时，这种事情经常发生。

  Groups/anti/maxandmin.sql

```
SELECT product_id, MAX(date_reported) AS latest,
  MIN(date_reported) AS earliest, bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

- 如果查询用到了两个不同的聚合函数，比如MAX()和MIN()，这可能会定位到一组中两条不同的记录。那这组应该返回哪个bug_id？

  Groups/anti/sumbyproduct.sql

```
SELECT product_id, SUM(hours) AS total_project_estimate,
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

这些就是为什么单值规则是如此重要的原因。不是每个不遵照这个规则的查询都会导致模棱两可的结果，但大多数都是这样。如果数据库能够区分有歧义和无歧义查询，并且只有在数据会导致歧义的情况下才返回错误就好了。但这样会导致程序的可靠性降低，这意味着同样的查询可能是合理的，也可能是不合理的，唯一的标准竟然是数据的状态！

15.3 如何识别反模式

对于大多数数据库来说，当输入一个违背了单值规则的查询时，会立刻返回给你一个错误。下面这些就是不同数据库返回的错误信息。

- Firebird 2.1:

  ```
  Invalid expression in the select list (not
  contained in either an 聚合函数 or the GROUP BY
  clause)
  ```

- IBM DB2 9.5:

  ```
  An expression starting with "BUG_ID" specified
  in a SELECT clause, HAVING clause, or ORDER BY
  clause is not specified in the GROUP BY clause
  or it is in a SELECT clause, HAVING clause, or
  ORDER BY clause with a column function and no
  GROUP BY clause is specified.
  ```

- Microsoft SQL Server 2008:

  ```
  Column 'Bugs.bug_id' is invalid in the select
  list because it is not contained in either an
  聚合函数 or the GROUP BY clause.
  ```

- MySQL 5.1，在设置`ONLY_FULL_GROUP` SQL模式之后禁止歧义查询。

  ```
  'bugs.b.bug_id' isn't in GROUP BY
  ```

- Oracle 10.2:

  ```
  not a GROUP BY expression
  ```

- PostgreSQL 8.3:

  ```
  column "bp.bug_id" must appear in the GROUP BY
  clause or be used in an 聚合函数
  ```

在SQLite和MySQL中，有歧义的列可能包含不可预测的和不可靠的数据。在MySQL中，返回的值是这一组结果中的第一条记录，其排序规

则是按照实际的物理存储顺序来定义的。SQLite的结果正好与MySQL相反，它返回最后一条记录。这两者的行为模式都没有文档描述，并且这两个数据库都不保证在后续版本中依旧这么执行。注意这些特征并且合理设计你的查询语句来避免歧义，是你所需要做的。

> **GROUP BY和DISTINCT**
>
> SQL支持另一个查询筛选关键字DISTINCT，它的作用是对查询结果进行去重操作，这样最终返回的每一行都是唯一的。比如，下面的这个查询返回谁在哪天提交过Bug，但每个人每天只返回一条记录：
>
> ```
> SELECT DISTINCT date_reported, reported_by FROM Bugs;
> ```
>
> 分组查询如果去掉所有的聚合函数，也能完成同样的事情。通过使用GROUP BY，也能减少到GROUP BY中的列的每一个不同组合都只返回一条查询结果：
>
> ```
> SELECT date_reported, reported_by FROM Bugs
> GROUP BY date_reported, reported_by;
> ```
>
> 这两个查询返回同样的结果，而且其优化和执行过程也应该类似，因此这个例子中唯一的不同大概就是偏好问题了。

15.4 合理使用反模式

正如我们所见的，MySQL和SQLite不能保证那些不满足单值规则的列返回的数据可靠。有些情况下你能利用这种不严谨的规则获得一些好处。

> Groups/legit/functional.sql

```
SELECT b.reported_by, a.account_name
FROM Bugs b JOIN Accounts a ON (b.reported_by = a.account_id)
GROUP BY b.reported_by;
```

在之前的查询中，account_name列从技术上来说违背了单值规则，因为它既没有出现在GROUP BY子句中，也没有经过聚合函数的处理。然而，每组中的account_name只可能有一个值，这个查询中的

分组是依照Bugs.reported_by来进行的，而这个列是指向Accounts表的一个外键，因此，分组规则满足和Accounts表的一对一关系。

换句话说，如果你知道reported_by的值，那就可以毫无歧义地断定对应的account_name，就好像你是根据Accounts表的主键进行查询一样。

这种没有歧义的关系叫做函数依赖。最常见的例子就是表的主键和对应的值：account_name和它的主键account_id之间是一个函数依赖。如果你对一张表的主键进行分组查询，那么每一个分组都会定位到表中唯一的一行，因此这一组中的其他列就必然只会有一个值。

Bugs.reported_by和Accounts表中的其他依赖属性有类似的关系，因为它引用了Accounts表的主键。当查询是基于reported_by的分组时，由于它是一个外键，Accounts表的属性是函数依赖的，那么查询结果不会产生任何歧义。

然而，大多数的数据库依然会返回一个错误。不仅因为SQL标准要求这样的行为，而且在执行中要找出这样的函数依赖的开销也不是很大[1]。但如果你使用的是MySQL或者SQLite并且小心谨慎地处理那些函数依赖列上的查询，你就能使用这样的查询并且避免歧义数据。

> [1] 本章的例子都很简单。要确认其他任意的SQL查询是不是功能依赖会比这些例子困难得多。

15.5　解决方案：无歧义地使用列

接下来的几节将介绍几种方法解决这个反模式，教你如何写出不带歧义的查询语句。

15.5.1　只查询功能依赖的列

最直接的解决方案就是将有歧义的列排除出查询。

> Groups/anti/groupbyproduct.sql

```
SELECT product_id, MAX(date_reported) AS latest
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

这个查询取出产品最新Bug提交的日期，尽管它并不获取最新Bug的bug_id。很多时候这就够了，别忽视简单的解决方案。

15.5.2　使用关联子查询

关联子查询会引用外联结查询，并且根据外联结查询中的每一条记录最终返回不同的结果。通过用子查询来搜索同一个产品中Bug日期的更新，可以找到每个产品最新的Bug。只要子查询没有返回，那么外联结查询到的Bug就是最新的。

Groups/soln/notexists.sql

```
SELECT bp1.product_id, b1.date_reported AS latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 USING (bug_id)
WHERE NOT EXISTS
  (SELECT * FROM Bugs b2 JOIN BugsProducts bp2 USING (bug_id)
   WHERE bp1.product_id = bp2.product_id
     AND b1.date_reported < b2.date_reported);
```

这个查询非常地简单易读。然而，要记住这个查询的性能并不是最好的，因为外联结查询结果中的每一条记录都会执行一遍关联的子查询。

15.5.3　使用衍生表

你可以使用衍生表来执行子查询，先得到一个临时的结果，只包含product_id和其对应的最新的Bug报告日期。然后使用这个临时表和原表进行联结查询，然后就能得到每个产品的最新的Bug信息。

Groups/soln/derived-table.sql

```
SELECT m.product_id, m.latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 USING (bug_id)
  JOIN (SELECT bp2.product_id, MAX(b2.date_reported) AS lates
   FROM Bugs b2 JOIN BugsProducts bp2 USING (bug_id)
   GROUP BY bp2.product_id) m
  ON (bp1.product_id = m.product_id AND b1.date_reported = m.
```

product_id	latest	bug_id
1	2010-06-01	2248

2	2010-02-16	3456
2	2010-02-16	5150
3	2010-01-01	5678

注意一点，如果子查询返回的最新日期匹配多个行，那么对应每个产品，你会获得多个行。如果你要确保每个product_id仅有一条记录，就可以在外联结查询时使用另一个分组函数：

Groups/soln/derived-table-no-duplicates.sql

```
SELECT m.product_id, m.latest, MAX(b1.bug_id) AS latest_bug_id
FROM Bugs b1 JOIN
  (SELECT product_id, MAX(date_reported) AS latest
   FROM Bugs b2 JOIN BugsProducts USING (bug_id)
   GROUP BY product_id) m
  ON (b1.date_reported = m.latest)
GROUP BY m.product_id, m.latest;
```

product_id	latest	latest_bug_id
1	2010-06-01	2248
2	2010-02-16	5150
3	2010-01-01	5678

衍生表方案是一个相对于关联子查询可扩展性更好的方案。衍生表并不是关联的，因此大多数品牌的数据库都能够一次执行子查询。然而，数据库必须将临时得到的记录存在一张临时表 中，因此，这个方案也不是性能最优的方案。

15.5.4 使用JOIN

你可以创建一个联结查询去匹配那些可能不存在的记录。这样的联结查询被称为外联结查询。如果尝试匹配的记录不存在，就会使用NULL来替代相应的列。因此，如果查询结果返回了NULL，我们就知道没有找到相应的记录。

Groups/soln/outer-join.sql

```
SELECT bp1.product_id, b1.date_reported AS latest, b1.bug_id
FROM Bugs b1 JOIN BugsProducts bp1 ON (b1.bug_id = bp1.bug_id)
```

```
LEFT OUTER JOIN (Bugs AS b2 JOIN BugsProducts AS bp2 ON (b2.b
  ON (bp1.product_id = bp2.product_id AND (b1.date_reported <
    OR b1.date_reported = b2.date_reported AND b1.bug_id < b2
WHERE b2.bug_id IS NULL;
```

product_id	latest	bug_id
1	2010-06-01	2248
2	2010-02-16	3150
3	2010-01-01	5678

对于大多数人来说，要看明白这个查询需要花点时间，并且需要在草稿纸上做点标记、计算。但是，一旦你弄明白它是怎么回事之后，它会变成一个很重要的工具。

JOIN解决方案适用于针对大量数据查询并且可伸缩性比较关键时。尽管这个方案比较难以理解和维护，但它总是能比基于子查询的解决方案更好地适应数据量的变化。记住一定要对不同类型的查询的性能进行实际的测量，而不是仅靠猜测来判断哪个更好。

15.5.5　对额外的列使用聚合函数

你可以通过对额外的列使用另一个聚合函数，从而使得它们遵从单值规则。

Groups/soln/extra-aggregate.sql

```
SELECT product_id, MAX(date_reported) AS latest,
  MAX(bug_id) AS latest_bug_id
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

只有确定最新的bug_id对应的Bug的日期也是最新的时候，才能使用这个方案，也就是说，Bug是按照时间顺序提交的。

15.5.6　连接同组所有值

最后，还有一个聚合函数可以用来处理bug_id并避免违背单值规则。MySQL和SQLite提供了一个叫做GROUP_CONCAT()的函数，它能将这一组中所有的值连在一起作为单一值返回，默认情况下，返回

的是一个由逗号分割的字符串。

Groups/soln/group-concat-mysql.sql

```
SELECT product_id, MAX(date_reported) AS latest
  GROUP_CONCAT(bug_id) AS bug_id_list,
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

product_id	latest	bug_id_list
1	2010-06-01	1234,2248
2	2010-02-16	3456,4077,5150
3	2010-01-01	5678,8063

这个查询并不会告诉你哪个bug_id对应最新的日期，bug_id_list包含了这一组中的所有bug_id。

这个方案的另一个缺点是，它并非SQL标准函数。其他的数据库并不支持这个函数。有些数据库支持自定义函数和自定义聚合函数。比如，PostgreSQL中可以这么处理：

Groups/soln/group-concat-pgsql.sql

```
CREATE AGGREGATE GROUP_ARRAY (
  BASETYPE = ANYELEMENT,
      SFUNC = ARRAY_APPEND,
      STYPE = ANYARRAY,
  INITCOND = '{}'
);

SELECT product_id, MAX(date_reported) AS latest
  ARRAY_TO_STRING(GROUP_ARRAY(bug_id), ',') AS bug_id_list,
FROM Bugs JOIN BugsProducts USING (bug_id)
GROUP BY product_id;
```

另一些数据库并不支持自定义函数，因此实现这个解决方案需要写存储过程遍历一个非分组的查询结果，手动地将每个值连在一起。

当你希望非Group By列在每一组中只有一个值，但实际存储的数据依旧违背单值规则的情况下，可以使用这种方案。

> 遵循单值规则，避免获得模棱两可的查询结果。

第16章　随机选择

> 随机数的产生实在太重要了，不能够让它由偶然性来决定。
>
> 罗伯特·科维欧

一个带广告系统的网站，需要在用户每次访问页面的时候随机选择一个广告来展示，让每一个广告投递商都有同等机会展示其广告，并且读者也不会因为每次都显示一样的广告而觉得无聊。

最开始的几天一切都好，但是逐渐地，网站变得越来越慢。几个星期以后，人们就开始抱怨网站太慢了。你通过测试真实页面的加载速度发现，这并不是心理作用。你的读者开始失去兴趣，网站流量也在降低。

根据过往的经验，你首先想到使用性能工具和一个带有示例数据的测试数据库来定位性能瓶颈。但是奇怪的是，通过测试页面加载速度，发现用来生成页面的每一个SQL查询看上去都没有问题。但生产环境上的网站的确正变得越来越慢。

最终，你意识到生产环境上的数据库要比你的测试样本大得多得多。于是你用了一个同等规模的数据库重新进行测试，发现问题出在广告选择的那个查询上。随着广告的数量越来越多，随机选择的性能直线下降。你已经发现了这个扩展性很差的查询，这就迈出了重要的第一步。

你要怎么在网站丢失所有的读者和赞助商之前，重新设计这个随机选择广告的查询？

16.1　目标：获取样本记录

执行随机返回结果的查询的频率远大于我们的预期。从一个大的数据集中返回数据样例是很平常的事情，这似乎和可重用、确定性的程序设计原则相违背。比如下面这些情况：

- 轮流展示的内容，比如广告或者推荐的新闻；
- 审核记录的子集；
- 给当前可用的操作对象指派输入请求；
- 生成测试数据。

相比于将整个数据集读入程序中再取出样例数据集，直接通过数据库查询拿出这些样例数据集会更好。

本章的目标就是要写出一个仅返回随机数据样本的高效SQL查询[1]。

> [1] 数学家和计算机科学家对真实随机和伪随机进行了详细的区分。在实际中，计算机只能产生伪随机数。

16.2　反模式：随机排序

获取随机记录的最常见SQL方法，就是对查询结果进行随机排序，然后获取第一行。这种技术理解起来很方便，实现起来也很方便：

> Random/anti/orderby-rand.sql

```
SELECT * FROM Bugs ORDER BY RAND() LIMIT 1;
```

尽管这是一个很流行的解决方案，但它的弱点也很鲜明。要理解这种方案的弱点，我们要先来和普通排序进行一下比较。一般的排序过程中，我们对某一列中的值进行两两比较，根据比较结果来排序行。这种排序方法是可复用的，当你执行多次这样的排序时，每次返回的结果都是一样的。同时这种方式也能受益于索引，因为索引本身就是根据给定的列排序的。

> Random/anti/indexed-sort.sql

```
SELECT * FROM Bugs ORDER BY date_reported;
```

如果排序的依据是给每一条记录分配一个随机值的函数，通过比较这个随机值来确定谁大谁小，那这样的结果就和记录本身的值无关，这样的排序每次执行的结果也都不一样。这正是目前我们想要的结果。

使用不定表达式（RAND()）意味着整个排序过程无法利用索引，因为没有索引会基于随机函数返回的值。这就是随机的作用：每次选择的时候都不同并且不可预测。

这就是影响查询性能的问题所在，因为使用索引是加速排序的最好方法。不使用索引的后果就是查询结果不得不由数据库“手动地”重新排序，这被称为一次全表遍历，并且经常伴随着将整个结果集保存到临时表中以及通过物理交换表内数据顺序操作来进行排序的情况。一次全表遍历排序比使用索引排序要慢很多，并且性能的差异随着数据量的增长而更加显著。

随机排序的另一个问题是，好不容易对整个数据集完成排序，但绝大多数的结果都浪费了，因为除了返回第一行之外，其他结果都立刻被丢弃了。在一个上千行的表中，为什么要费力地随机排序所有的记录，而我们所需要的仅仅是其中的一行？

这些问题在数据量很小的时候都是不容易被发现的，因此在开发和测试过程中它都可能是一个很好的方案。但随着数据量不断地增长，这个查询却无法很好地扩展。

16.3　如何识别反模式

在16.2节中描述的这个技术非常简单，并且很多程序员可能是在某篇文章中看到过或者自己研究过，都在使用这项技术。下面的这些问题可能就是你的同事用了本章的反模式的线索。

- “在SQL中，返回一个随机行真慢啊！”

  在开发和测试环境中，由于数据量较小，获取随机样本的查询语句工作得很好。但随着真实数据的增长，它也逐渐变得缓慢。没什么数据库调优方案、索引或者缓存能够提升它的扩展性。

- “我要怎么增加我的程序的可使用内存大小？我要获取所有的记录然后随机选择一个。”

  你不应该将所有数据载入到程序内存中，这么做非常浪费资源。除此之外，数据库的数据会不断增长直到程序内存完全不够用。

- “你是不是也觉得有些列出现的频率比别的要高一些？这个随机

算法貌似不是很随机。”

数据主键并不连续，并且随机数生成算法并没有考虑到这一点（参考16.5节）。

16.4　合理使用反模式

随机排序的性能问题在数据量很小的时候是可容忍的。

比如，你可以随机选择程序员去处理一个指定的Bug。我们可以保证，程序员的数量永远不会大到需要使用高扩展性随机算法。

另一个例子就是从美国50个州中随机选择一个，这个列表的大小适度，而且在我们有生之年不太可能改变。

16.5　解决方案：没有具体的顺序……

随机选择是需要全表遍历并且耗时地进行手动排序的一个典型案例。在你设计SQL查询时，应该仔细地检查是否也有类似的查询。与其徒劳地想办法优化一个不可被优化的查询，还不如考虑别的实现方案。你可以使用接下来几段中所介绍的方法来获得一条随机记录。与随机排序不同的是，下面的每一个方案都能高效地获得同样的随机效果。

16.5.1　从1到最大值之间随机选择

一种避免对所有数据进行排序的方法，就是在1到最大的主键值之间随机选择一个。

Random/soln/rand-1-to-max.sql

```
SELECT b1.*
FROM Bugs AS b1
JOIN (SELECT CEIL(RAND() * (SELECT MAX(bug_id) FROM Bugs)) AS
  ON (b1.bug_id = b2.rand_id);
```

这个方案假设主键的值是从1开始并且保持连续。这意味着在1到最大值之间没有任何值是未使用的。如果当中漏掉一些值，那随机获得的主键可能取不到任何数据。

当确信主键是从1到最大值连续的时候，可以使用这个方案。

16.5.2　选择下一个最大值

这个方案和前一个方案类似，但解决了在1到最大值之间有缝隙的情况，这个查询会返回它随机找到的第一个有效的值。

Random/soln/next-higher.sql

```
SELECT b1.*
FROM Bugs AS b1
JOIN (SELECT CEIL(RAND() * (SELECT MAX(bug_id) FROM Bugs)) AS
WHERE b1.bug_id >= b2.bug_id
ORDER BY b1.bug_id
LIMIT 1;
```

这个方法解决了随机数没有主键的情况，同时也意味着在一个缝隙之后的那个值被选中的概率会增大。即使在一个离散的队列中，随机也应该保证每个值出现的概率相等，但这里的bug_id并不是如此。

当队列中的缝隙不大并且每个值要被等概率选择的重要性不高时，可以考虑使用这种方案。

16.5.3　获取所有的键值，随机选择一个

你可以使用程序代码来获取所有的主键值，然后随机选择一个。再使用这个随机选择出来的主键查询完整的记录。可以用如下的PHP代码实现：

Random/soln/rand-key-from-list.php

```
<?php
$bug_id_list = $pdo->query("SELECT bug_id FROM Bugs")->fetchA

$rand = random( count($bug_id_list) );
$rand_bug_id = $bug_id_list[$rand]["bug_id"];
$stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id = ?");
$stmt->execute( array($rand_bug_id) );
$rand_bug = $stmt->fetch();
```

这个方法避免了对全表的排序，并且选择每个键的概率相同，但它也有其他的开销。

- 获取所有的bug_id时会得到一个过长的列表，可能会超出程序所能处理的内存极限，并且导致如下的错误：

```
Fatal error: Allowed memory size of 16777216 bytes exhau
```

- 查询必须执行两次：一次获取主键的列表，第二次获取对应的记录。如果查询太复杂或者太耗时，这就会成为问题。

当查询逻辑很简单并且数据量适度的时候，可以考虑使用这个方案。这个方案在处理非连续值时效果很好。

16.5.4 使用偏移量选择随机行

还有另一个方案来避免之前几个方案中的问题，那就是计算总的数据行数，随机选择0到总行数之间的一个值，然后用这个值作为位移来获取随机行。

Random/soln/limit-offset.php

```
<?php

$rand = "SELECT ROUND(RAND() * (SELECT COUNT(*) FROM Bugs))";
$offset = $pdo->query($rand)->fetch(PDO::FETCH_ASSOC);

$sql = "SELECT * FROM Bugs LIMIT 1 OFFSET :offset";
$stmt = $pdo->prepare($sql);
$stmt->execute( $offset );
$rand_bug = $stmt->fetch();
```

这个方案使用了非标准的LIMIT子句，MySQL、PostgreSQL和SQLite支持这一子句。

Oracle、Microsoft SQL Server和IBM DB2使用另一个叫做ROW_NUMBER()的函数。

比如，下面是Oracle的解决方案：

Random/soln/row_number.php

```
<?php
$rand = "SELECT 1 + MOD(ABS(dbms_random.random()),
  (SELECT COUNT(*) FROM Bugs)) AS offset FROM dual";
$offset = $pdo->query($rand)->fetch(PDO::FETCH_ASSOC);

$sql = "WITH NumberedBugs AS (
  SELECT b.*, ROW_NUMBER() OVER (ORDER BY bug_id) AS RN FROM B
) SELECT * FROM NumberedBugs WHERE RN = :offset";
$stmt = $pdo->prepare($sql);
$stmt->execute( $offset );
$rand_bug = $stmt->fetch();
```

当你不能保证主键是连续的，并且需要每行都有相同的选中概率时，可以用这个方案。

16.5.5 专有解决方案

每种数据库都可能针对这个需求提供独有的解决方案，比如Microsoft SQL Server 2005增加了一个TABLE-SAMPLE子句：

Random/soln/tablesample-sql2005.sql

```
SELECT * FROM Bugs TABLESAMPLE (1 ROWS);
```

Oracle使用了一个类似的SAMPLE子句，比如返回表中1%的记录：

Random/soln/sample-oracle.sql

```
SELECT * FROM (SELECT * FROM Bugs SAMPLE (1)
ORDER BY dbms_random.value) WHERE ROWNUM = 1;
```

你应该仔细阅读所使用的数据库的说明文档，来找这些专有解决方案。通常都有一些限制或者额外参数需要学习。

> 有些查询是无法优化的，换种方法试试看。

第17章　可怜人的搜索引擎

> 有些人遇到问题时总是会说：“我知道，我会使用正则表达式。”然后他会碰到更多问题。
>
> 杰米·加文斯基

1995年，我在一家公司做技术支持，那时公司开始通过Web向客户提供信息。我们有一系列简短文档描述了常见问题的解决方案，我们想把这些文章放到Web做成一个知识库。

我们很快就意识到，随着文章数量的增长，知识库需要支持搜索功能，因为客户不想浏览几百篇文章才找到他们需要的解决方案。一个临时的应对方案是将文章分类，但即使如此，每个分类下的文章数量也很大，并且很多文章都可能属于多个分类。

我们想要让客户能够通过搜索文章，缩小候选列表。最灵活和直接的界面就是允许客户输入任意的关键字，然后给出包含这些关键字的文章。一篇文章如果和用户输入的关键字匹配度越高，则出现的越靠前。同时，我们也希望能够匹配词形变化。比如，对crash的搜索也要同时匹配crashed、crashes和crashing。当然这个搜索要能够在一个不断增长的文章集合中快速地返回结果，这样才能在一个Web程序中使用它。

如果上面这些细节描述让对你觉得多余，我并不感到惊讶。搜索技术在如今已经变得如此平常，以至于我们都回想不起来它出现之前的情况了。但是用SQL搜索关键字，同时保证快速和精确，依旧是相当地困难。

17.1　目标：全文搜索

任何存储文本的应用都有针对这个文本进行单词或词组搜索的需求。我们使用数据库存储越来越多的文本数据，同时也需要搜索速度越来越快。Web应用尤其需要高性能和高扩展性数据库搜索技术。

SQL的一个基本原理（以及SQL所继承的关系原理）就是一列中的单个数据是原子性的。也就是说，你能对两个值进行比较，但通常是把

两个值当成两个整体来比较。在SQL中比较子字符串总是意味着低效和不精确。

尽管如此，我们仍旧需要一个方法来比较一个较短的字符串和一个较长的字符串，并且查看是否较短字符串是较长字符串的一个子串。我们要怎么使用SQL来实现这样的需求呢？

17.2 反模式：模式匹配断言

SQL提供了模式匹配断言来比较字符串，并且这是很多程序员用来搜索关键字的第一选择。最广泛支持的就是LIKE断言。

LIKE断言提供了一个通配符（%）用以匹配0个或者多个字符。在一个关键字之前及之后使用通配符能够匹配到包含这个关键字的任意字符串。第一个通配符匹配了关键字之前的所有文本，第二个通配符匹配了关键字之后的所有文本。

Search/anti/like.sql

```
SELECT * FROM Bugs WHERE description LIKE '%crash%';
```

正则表达式也被很多数据库所支持，尽管这不是标准。你不需要使用通配符，因为基本上正则表达式能对所有子串进行模式匹配。如下是一个使用MySQL的正则表达式的例子[1]：

Search/anti/regexp.sql

```
SELECT * FROM Bugs WHERE description REGEXP 'crash';
```

[1] 尽管SQL-99标准定义了SIMILAR TO这个断言用来匹配正则表达式，但大多数SQL数据库还是使用非标准的语法。

使用模式匹配操作符的最大缺点就在于性能问题。它们无法从传统的索引上受益，因此必须进行全表遍历。由于对一个字符串列进行匹配操作非常耗时（相对来说，和比较两个整数是否相等所耗的时间相比），全表遍历所花的总时间就非常多。

使用LIKE或者正则表达式进行模式匹配搜索的另一个问题就是，经

常会返回意料之外的结果。

Search/anti/like-false-match.sql

```
SELECT * FROM Bugs WHERE description LIKE '%one%';
```

上面的这个例子是要匹配包含one这个单词的文本，同时也匹配到单词money、prone、lonely等。在关键字两端都加上空格也不能完美解决这个问题，因为无法匹配到后面直接跟着标点符号的单词或者正好在文本开头或结尾的单词。数据库支持的正则表达式可能会为单词边界提供一个模式来解决单词匹配问题[2]：

Search/anti/regexp-word.sql

```
SELECT * FROM Bugs WHERE description REGEXP '[[:<:]]one[[:>:]
```

[2] 本例使用的是MySQL的语法。

考虑到性能和扩展性的问题，以及预防不合理的匹配所做的工作，简单的模式匹配对于关键字搜索来说是一个糟糕的技术方案。

17.3 如何识别反模式

如下的一些问题通常预示着项目中使用了“可怜人的搜索引擎”这个反模式。

- “我要怎么在`LIKE`表达式的两个通配符之间插入一个变量？”

  通常当程序员想要使用模式匹配对用户输入的关键字进行搜索时，会产生这个问题。

- “我要怎么写一个正则表达式来检查一个字符串是否包含多个单词、不包含一个特定的单词，或者包含给定单词的任意形式？”

  如果一个复杂的问题看起来用正则表达式很难解决，那就是用了反模式了。

- “我们网站的搜索功能在加了很多文档进去之后慢得不可理喻了。出什么问题了？”

随着数据的增长，这个反模式方案就暴露出了它脆弱的扩展性问题。

17.4　合理使用反模式

在17.2节中所使用的SQL语句都是合法的，并且很简单直接。这意味着很多东西。

性能总是非常重要的，但一些查询过程很少执行，因此不需要花很多功夫对它进行优化。为一个很少使用的查询维护一个索引，可能就和用不高效的方法执行查询一样耗资源。如果这个查询是一个临时的查询，没人能保证你定义的索引能够给这个查询带来任何好处。

使用模式匹配操作进行复杂查询是很困难的，但如果你是为了一些简单的需求设计这样的模式匹配，它们就能帮助你用最少的工作量获得正确的结果。

17.5　解决方案：使用正确的工具

最好的方案就是使用特殊的搜索引擎技术，而不是SQL。另一个可选方案是将结果保存起来从而减少重复的搜索开销。

接下来的几节介绍了一些数据库的内置扩展和独立项目所提供的搜索技术。同时，我们也会设计一个完全使用SQL标准的解决方案，它显然会比使用子串匹配更高效。

17.5.1　数据库扩展

每个大品牌的数据库都有对全文搜索这个需求的解决方案，但这些方案并没有任何的标准，各个数据库的实现也互不兼容。如果只使用一种数据库品牌（或者打算使用开发商开发的特性），这些特性是获得高性能文本搜索的最佳途径，并且能和SQL查询整合得非常好。

如下是对一些SQL数据库的全文搜索技术的简要描述。具体的细节不断随数据库的版本升级而改变，因此记得要阅读一下对应当前使用版本的文档。

MySQL中的全文索引

MySQL为MyISAM存储引擎提供了一个简单地全文索引类型。你可以

在一个类型为CHAR、VARCHAR或者TEXT的列上定义一个全文索引。下面这个例子在Bug的summary和description列上定义了全文索引：

Search/soln/mysql/alter-table.sql

```
ALTER TABLE Bugs ADD FULLTEXT INDEX bugfts (summary, description);
```

可以使用MATCH()函数对索引内容进行搜索。必须在匹配时指定需要全文索引的列（因而你可以在同一张表中对其他列使用不同类型的索引）。

Search/soln/mysql/match.sql

```
SELECT * FROM Bugs WHERE MATCH(summary, description) AGAINST
```

自MySQL 4.1开始，你还能在表达式上使用简单的布尔运算来更精确地过滤查询结果。

Search/soln/mysql/match-boolean.sql

```
SELECT * FROM Bugs WHERE MATCH(summary, description)
  AGAINST ('+crash -save' IN BOOLEAN MODE);
```

Oracle中的文本索引

从1997年的Oracle 8开始，Oracle就支持了文本索引特性，当时它是数据资料夹的一部分，称为ConText。这项技术在随后的版本中多次更新，现在已经被集成到数据库程序里了。Oracle中的文本索引技术复杂且强大，因此这里只能非常简单地总结一下。

- CONTEXT

  为单个文本列建立一个这样的索引类型，使用CONTAINS()操作符进行搜索。索引和数据不保持同步，因此你每次都得手动重建索引或者定期重建。

  Search/soln/oracle/create-index.sql



```
CREATE INDEX BugsText ON Bugs(summary) INDEXTYPE IS CTSS

SELECT * FROM Bugs WHERE CONTAINS(summary, 'crash') > 0;
```

- CTXCAT

  这个类型的索引是针对短文本设计的，比如用在在线程序的分类目录上，和同一张表的其他结构化列一起。这种类型的索引会保持数据的同步。

  Search/soln/oracle/ctxcat-create.sql

```
CTX_DDL.CREATE_INDEX_SET('BugsCatalogSet');
CTX_DDL.ADD_INDEX('BugsCatalogSet', 'status');
CTX_DDL.ADD_INDEX('BugsCatalogSet', 'priority');

CREATE INDEX BugsCatalog ON Bugs(summary) INDEXTYPE IS C
  PARAMETERS('BugsCatalogSet');
```

  CATSEARCH()操作符分别接受两个参数来搜索索引列和结构化列的集合。

  Search/soln/oracle/ctxcat-search.sql

```
SELECT * FROM Bugs
WHERE CATSEARCH(summary, '(crash save)', 'status = "NEW"
```

- CTXXPATH

  这种类型的索引是针对XML文档搜索而设计的，使用existsNode()操作符。

  Search/soln/oracle/ctxxpath.sql

```
CREATE INDEX BugTestXml ON Bugs(testoutput) INDEXTYPE IS


SELECT * FROM Bugs
WHERE testoutput.existsNode('/testsuite/test[@status="fa
```

- CTXRULE

  假设在数据库中存了大量的文档，然后需要根据文档内容进行分类。

  使用CTXRULE索引，你可以设计分析文档的规则并返回文档的分类信息。同时，你可以提供一些示例文档以及对应的分类概念，然后让Oracle去分析这个规则并应用到其他文档上去。你甚至可以完全让这个过程自动化，让Oracle去分析文档结构然后自动分类。

  如何使用CTXRULE不在本书的讨论范围内。

Microsoft SQL Server的全文搜索

SQL Server 2000及后续版本支持全文索引，它的配置相对复杂，包括了语言、同义词库和自动数据同步选项。SQL Server提供了一系列的存储过程来创建全文索引，可以使用CONTAINS()操作符来使用全文索引。

要执行对包含crash的Bug进行搜索的例子，首先要做的就是启用全文特性，然后在数据库中定义一个目录：

Search/soln/microsoft/catalog.sql

```
EXEC sp_fulltext_database 'enable'
EXEC sp_fulltext_catalog 'BugsCatalog', 'create'
```

接着，为Bugs表定义一个全文索引，将列加入到索引中，然后激活索引：

Search/soln/microsoft/create-index.sql

```
EXEC sp_fulltext_table 'Bugs', 'create', 'BugsCatalog', 'bug_
EXEC sp_fulltext_column 'Bugs', 'summary', 'add', '2057'
EXEC sp_fulltext_column 'Bugs', 'description', 'add', '2057'
EXEC sp_fulltext_table 'Bugs', 'activate'
```

启用全文索引的自动同步功能，从而对索引列的操作会同时影响到索引，然后就可以开始填充索引。这会在后台执行，因而需要花点时间才能使用索引。

Search/soln/microsoft/start.sql

```
EXEC sp_fulltext_table 'Bugs', 'start_change_tracking'
EXEC sp_fulltext_table 'Bugs', 'start_background_updateindex'
EXEC sp_fulltext_table 'Bugs , 'start_full'
```

最后，使用`CONTAINS()`操作符执行查询：

Search/soln/microsoft/search.sql

```
SELECT * FROM Bugs WHERE CONTAINS(summary, '"crash"');
```

PostgreSQL的文本搜索

PostgreSQL 8.3提供了一个复杂的可大量配置的方式，来将文本转化为可搜索的词汇集合，并且让这些文档能够进行模式匹配搜索。

为了最大地提升性能，你需要将内容存两份：一份为原始文本格式，另一份为特殊的`TSVECTOR`类型的可搜索格式。

Search/soln/postgresql/create-table.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY,
  summary      VARCHAR(80),
  description  TEXT,
  ts_bugtext   TSVECTOR
  -- other columns
);
```

你需要确保`TSVECTOR`列的内容和你所想要搜索的列的内容同步。

PostgreSQL提供了一个内置的触发器来简化这一操作：

Search/soln/postgresql/trigger.sql

```
CREATE TRIGGER ts_bugtext BEFORE INSERT OR UPDATE ON Bugs
FOR EACH ROW EXECUTE PROCEDURE
  tsvector_update_trigger(ts_bugtext, 'pg_catalog.english', s
```

你也应该同时在TSVECTOR列上创建一个反向索引（GIN）：

Search/soln/postgresql/create-index.sql

```
CREATE INDEX bugs_ts ON Bugs USING GIN(ts_bugtext);
```

在做完这一切之后，就可以在全文索引的帮助下使用PostgreSQL的文本搜索操作符@@来高效地执行搜索查询：

Search/soln/postgresql/search.sql

```
SELECT * FROM Bugs WHERE ts_bugtext @@ to_tsquery('crash');
```

有很多其他的选项来自定义可搜索的内容、搜索查询和搜索结果。

SQLite的全文搜索（FTS）

SQLite中的标准表结构并不支持高效的全文搜索，但你可以使用SQLite的一个可选扩展组件来存储可搜索的文本。这些内容会存储在一个特殊的虚拟表结构中。它有三个版本的扩展，FTS1、FTS2和FTS3。

FTS扩展在标准的SQLite编译版本中默认是未启用的，因此你需要修改编译选项来启用FTS，并重新编译SQLite。比如，在Makefile.in中增加如下内容，然后编译SQLite。

Search/soln/sqlite/makefile.in

```
TCC += -DSQLITE_CORE=1
TCC += -DSQLITE_ENABLE_FTS3=1
```

一旦你有了一个启用FTS的SQLite版本，就可以创建一个虚拟表来存储可搜索文本。任何数据类型、约束或者其他列选项将在搜索时被忽略。

Search/soln/sqlite/create-table.sql

```
CREATE VIRTUAL TABLE BugsText USING fts3(summary, description
```

如果你在为另一张表建立索引（比如这个例子中的`Bugs`表），就必须将数据复制到虚拟表中。FTS的虚拟表默认会包含一个主键列，叫做docid，因此可以将原表中的行关联起来。

Search/soln/sqlite/insert.sql

```
INSERT INTO BugsText (docid, summary, description)
  SELECT bug_id, summary, description FROM Bugs;
```

现在你可以对FTS的虚拟表`BugsText`进行查询，使用一个高效的全文搜索断言`MATCH`，然后把匹配的行和原表`Bugs`进行联结。将`FTS`表的名字当成一个虚拟的列来进行针对任意列的模式匹配。

Search/soln/sqlite/search.sql

```
SELECT b.* FROM BugsText t JOIN Bugs b ON (t.docid = b.bug_id
WHERE BugsText MATCH 'crash';
```

匹配条件同时也支持功能有限的布尔表达式。

Search/soln/sqlite/search-boolean.sql

```
SELECT * FROM BugsText WHERE BugsText MATCH 'crash -save';
```

17.5.2　第三方搜索引擎

如果需要搜索功能的代码对不同的数据库都通用，那就需要一个独立于SQL数据库的搜索引擎。这一节简要介绍两个产品：Sphinx Search和Apache Lucene。

Sphinx Search

Sphinx Search（http://www.sphinxsearch.com/）是一个开源的搜索引擎，用于和MySQL及PostgreSQL配套使用。在本书出版时，一个非官方的Sphinx Search补丁支持开源的Firebird数据库。可能在将来的版本中，这个搜索引擎会考虑支持其他的数据库。

在Sphinx Search中，构建索引和搜索都很快，而且它也支持分布式查询。对于数据不常更新且要求高可扩展性的程序来说，Sphinx Search是个很好的选择。

你可以使用Sphinx Search来索引存在于MySQL数据库中的数据。通过修改`sphinx.conf`中的几个字段，你可以指定对应的数据库。你必须写一些SQL查询脚本为构建索引的操作获取数据。这个查询要求第一列为一个整型主键。你可以声明一些属性列来对结果进行约束或者排序。余下的列就是用来进行全文索引的。最后，另一个SQL查询脚本通过给定的主键代码`$id`，从数据库中获取完整的记录。

Search/soln/sphinx/sphinx.conf

```
source bugsrc
{
  type                   = mysql
  sql_user               = bugsuser
  sql_pass               = xyzzy
  sql_db                 = bugsdatabase
  sql_query              = \
    SELECT bug_id, status, date_reported, summary, descriptio
    FROM Bugs
  sql_attr_timestamp     = date_reported
  sql_attr_str2ordinal = status
  sql_query_info         = SELECT * FROM Bugs WHERE bug_id = $i
}

index bugs
{
  source                 = bugsrc
  path                   = /opt/local/var/db/sphinx/bugs
}
```

在配置完成`sphinx.conf`之后，你就可以在shell中使用`indexer`命令创建索引了：

Search/soln/sphinx/indexer.sh

```
indexer -c sphinx.conf bugs
```

你可以使用search命令对索引进行搜索：

Search/soln/sphinx/search.sh

```
search -b "crash -save"
```

Sphinx Search也有一个守护进程以及对应的API给常用的脚本语言（诸如PHP、Perl和Ruby）使用。当前版本的最主要问题是，目前的索引算法不支持高效的增量更新。在一个经常更新的数据源上使用Sphinx Search需要一些折中的处理办法。比如说，将可搜索表拆分成两张表，第一张表存储不变的主体历史数据，第二张表存储一个较小的当前数据的集合。当数据逐渐增长的时候，就需要重建索引。然后你的程序必须对两个Sphinx Search索引进行搜索。

Apache Lucene

Lucene（http://lucene.apache.org/）是一个针对Java程序的成熟搜索引擎。还有一些类似的项目，只是所使用的语言不同，包括C++、C#、Perl、Python、Ruby和PHP。

Lucene使用其独有的格式为文本文档创建索引。Lucene索引不和元数据保持同步。如果你插入、删除或者更新数据库中的数据，必须同时也对Lucene索引进行对应的操作。

使用Lucene搜索引擎有点像使用一台汽车发动机，必须要有一堆相关技术支持才能让它工作。Lucene不会直接从SQL数据库中读取数据集合，你必须手动往Lucene索引中写入文档。比如，你可以执行一个SQL查询，然后对结果中的每一行，创建一个Lucene文档对象然后保存到Lucene索引中。你可以通过Lucene的Java API来使用对应的功能。

所幸的是，Apache提供了另一个项目叫做Solr（http://lucene.apache.org/solr/）。Solr是一个Lucene索引的网关服务。你可以向Solr添加文档或者使用一个REST风格的接口提交查询请求，然后就可以使用任意的语言来调用Lucene的服务了。

你也可以将Solr配置成直接连接到数据库，执行一个查询操作，然后使用DataImportHandler工具来对结果进行索引。

实现自己的搜索引擎

假设你不想使用不同数据库特有的搜索特性，也不想安装一个独立的搜索引擎产品。你需要一个高效的、与数据库品牌无关的解决方案来进行文本搜索。在本节中，我们将使用一个称为反向索引的方案。基本上来说，反向索引就是一个所有可能被搜索的单词列表。在多对多的关系中，索引将这些单词和包含这些单词的文本关联起来。也就是说，crash这个单词出现在很多Bug描述中，然后每个Bug描述又包含很多别的关键字。这一节就来介绍如何设计反向索引。

首先，定义一张Keywords表来记录所有用户用来搜索的关键字，然后定义个交叉表BugsKewords来建立多对多的关系：

Search/soln/inverted-index/create-table.sql

```
CREATE TABLE Keywords (
  keyword_id  SERIAL PRIMARY KEY,
  keyword     VARCHAR(40) NOT NULL,
  UNIQUE KEY (keyword)
);

CREATE TABLE BugsKeywords (
  keyword_id  BIGINT UNSIGNED NOT NULL,
  bug_id      BIGINT UNSIGNED NOT NULL,
  PRIMARY KEY (keyword_id, bug_id),
  FOREIGN KEY (keyword_id) REFERENCES Keywords(keyword_id),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id)
);
```

接下来，将每个关键字和其所匹配到的Bug添加到BugsKeywords表中。我们可以使用LIKE或者正则表达式来执行子字符串匹配查询，获得我们所需要的匹配记录。这种方式不比在17.2节中所说的查询有更多的开销，但由于我们只执行一次这样的查询，因此省下了很多时间。如果我们将查询结果存储在交叉表中，所有之后对同一个关键字的搜索都会快很多。

接下来，我们写一个存储过程来简化对一个给定关键字的搜索[3]。如果给定的关键字已经被搜索过了，这个查询就会很快，因为

BugsKeywords表中的记录就是包含这个给定的关键字的文章列表。如果还没人搜索过这个给定的关键字，我们就需要使用原始的方法对所有的文本记录进行全文搜索。

3 这个例子使用MySQL的语法来编写存储过程。

Search/soln/inverted-index/search-proc.sql

```
CREATE PROCEDURE BugsSearch(keyword VARCHAR(40))
BEGIN
  SET @keyword = keyword;
①     PREPARE s1 FROM 'SELECT MAX(keyword_id) INTO @k FROM Ke
        WHERE keyword = ?';
      EXECUTE s1 USING @keyword;
      DEALLOCATE PREPARE s1;

      IF (@k IS NULL) THEN
②       PREPARE s2 FROM 'INSERT INTO Keywords (keyword) VALUE
        EXECUTE s2 USING @keyword;
        DEALLOCATE PREPARE s2;
③       SELECT LAST_INSERT_ID() INTO @k;

④       PREPARE s3 FROM 'INSERT INTO BugsKeywords (bug_id, ke
          SELECT bug_id, ? FROM Bugs
          WHERE summary REGEXP CONCAT(''[[:<:]]'', ?, ''[[:>
            OR description REGEXP CONCAT(''[[:<:]]'', ?, ''[
        EXECUTE s3 USING @k, @keyword, @keyword;
        DEALLOCATE PREPARE s3;
      END IF;

⑤     PREPARE s4 FROM 'SELECT b.* FROM Bugs b
        JOIN BugsKeywords k USING (bug_id)
        WHERE k.keyword_id = ?';
      EXECUTE s4 USING @k;
      DEALLOCATE PREPARE s4;
    END
```

① 搜索用户指定关键字。从keywords、keyword_id或NULL返回整型主键，如果这个词之前未出现过。

② 如果未找到该词，将它当做新词插入。

③ 查询keywords生成的主键值。

④ 通过搜索有新关键字的Bug来填入交叉表。

⑤ 最后，查询符合keyword_id的整行，无论这个关键字存在或者需要当做新词条插入。

现在我们可以执行这个存储过程，然后传入想要的关键字。这个存储过程会返回匹配到的 Bug，不论它是需要搜索全表来找到匹配的Bug然后追加到交叉表中，还是直接就可以从交叉表中获取相应的结果。

Search/soln/inverted-index/search-proc.sql

```
CALL BugsSearch('crash');
```

这个方案还有一个需要注意的是：在有新文档添加到数据库中时，我们需要定义一个触发器以填充交叉表。如果你需要支持对Bug描述的更新操作，还要写一个触发器去重新分析文本，然后添加或删除BugsKeywords表中的记录。

Search/soln/inverted-index/trigger.sql

```
CREATE TRIGGER Bugs_Insert AFTER INSERT ON Bugs
FOR EACH ROW
BEGIN
  INSERT INTO BugsKeywords (bug_id, keyword_id)
    SELECT NEW.bug_id, k.keyword_id FROM Keywords k
    WHERE NEW.description REGEXP CONCAT('[[:<:]]', k.keyword,
           OR NEW.summary REGEXP CONCAT('[[:<:]]', k.keyword,
END
```

这个关键字列表会随着用户不断地搜索而自动增长，因此我们不必将每个在知识库中找到的单词都加到列表中去。从另一个角度来说，如果我们可以预测一些关键字，就可以简单地在程序初始化的时候执行一下这几个关键字的搜索。这样，这部分的时间开销就不会由用户来承担了。

在本章最开始的知识库的例子中我使用了反向索引，我还在

Keywords表中添加了额外的一列num_searches。这一列在每次这个关键字被搜索时会自增，我使用这一列来跟踪搜索关键字的分布。

> 你不必使用SQL来解决所有问题。

第18章　意大利面条式查询

> 实体不应不必要地增殖。
>
> 奥卡姆

你的老板正和他的老板打电话，然后他示意你过去。他用手遮住了话筒，小声对你说："执行委员会在开预算会议，我们可能要被裁员，除非我能有数据告诉副总裁我们的部门一直都很忙。我需要知道我们同时在开发多少个项目，多少个程序员在修补Bug，每个程序员平均修补多少个Bug，以及多少修复了的Bug是由用户报告的。现在就要！"

你飞奔至座位，打开SQL工具，开始写查询语句。你想要立刻得到所有的数据，于是你写了一个很复杂的查询脚本，希望能尽可能减少重复的工作量，能更快地得到数据。

> Spaghetti-Query/intro/report.sql

```
SELECT COUNT(bp.product_id) AS how_many_products,
  COUNT(dev.account_id) AS how_many_developers,
  COUNT(b.bug_id)/COUNT(dev.account_id) AS avg_bugs_per_devel
  COUNT(cust.account_id) AS how_many_customers
FROM Bugs b JOIN BugsProducts bp ON (b.bug_id = bp.bug_id)
JOIN Accounts dev ON (b.assigned_to = dev.account_id)
JOIN Accounts cust ON (b.reported_by = cust.account_id)
WHERE cust.email NOT LIKE '%@example.com'
GROUP BY bp.product_id;
```

数据出来了，但看上去是错的。我们怎么会有几十个产品？怎么可能平均每个程序员正好修复1.0个Bug？你的老板要的不是客户数量，他要得是客户报告的Bug数量。这些数据怎么会是这样呢？这个查询比你所想的要更加复杂。

你的老板挂掉了电话。“无所谓了，”他叹息道，“太晚了。我们收拾下桌子吧。”

18.1　目标：减少SQL查询数量

SQL开发人员经常会被同一个问题困扰：“我要怎么用一个查询来完成这件事情？”这个问题基本上在任务中都会被提及。受过培训的程序员认为，一个SQL查询是困难的、复杂的和耗资源的，那么两个SQL查询就是糟糕度乘以二。用多于两个的SQL查询来解决问题根本不在考虑范围内。

程序员不能减少他们任务的复杂度，但他们想要简单化其解决方案。他们使用“优雅”或者“高效”这些形容词来描述他们的目标，并且认为使用一条SQL查询就能完成目标。

18.2　反模式：使用一步操作解决复杂问题

SQL是一门极具表现力的语言——你可以在单个SQL查询或者单条语句中完成很多事情。但这并不意味着必须强制只使用一行代码，或者认为使用一行代码就搞定每个任务是个好主意。你在使用其他语言的时候也有这样的习惯吗？应该没有吧。

18.2.1　副作用

通过一个查询来获得所有结果的常见后果就是得到了一个笛卡儿积。当查询中的两张表之间没有条件限制其关系时，就会发生这样的情况。没有对应的限制而直接使用两张表进行联结查询，就会得到第一张表中的每一行和第二张表中的每一行的一个组合。每一个这样的组合就会成为结果集中的一行，最终你就得到一个行数多很多的结果集。

我们来看个例子。假设我们想要查询`Bugs`数据库，计算一个给定的产品有多少Bug被修复了，多少Bug正处于打开状态。很多程序员可能会写出如下的这条语句：

Spaghetti-Query/anti/cartesian.sql

```
SELECT p.product_id,
  COUNT(f.bug_id) AS count_fixed,
```

```
  COUNT(o.bug_id) AS count_open
FROM BugsProducts p
LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status =
LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status =
WHERE p.product_id = 1
GROUP BY p.product_id;
```

你碰巧知道，事实上对于给定的这个产品，有12个Bug被修复了，有7个Bug是打开的。因此，结果看上去很耐人寻味：

product_id	count_fixed	count_open
1	84	84

是什么导致结果和预期相差十万八千里？没那么巧，正好是84 = 12×7。这个例子将`Products`表和两个不同的`Bugs`表的子集联合起来，结果却是那两个子集的笛卡儿积。12个FIXED状态的Bug中的每一个和一个OPEN状态的Bug凑成了一对。

你可以想象，笛卡儿积和图18-1所画的一样。每条线链接了一个已修复的Bug和一个打开的Bug，然后成为了临时结果集中的一行（在分组语句执行之前）。我们可以注释掉`GROUP BY`子句和那些聚合函数来查看这个查询的结果。

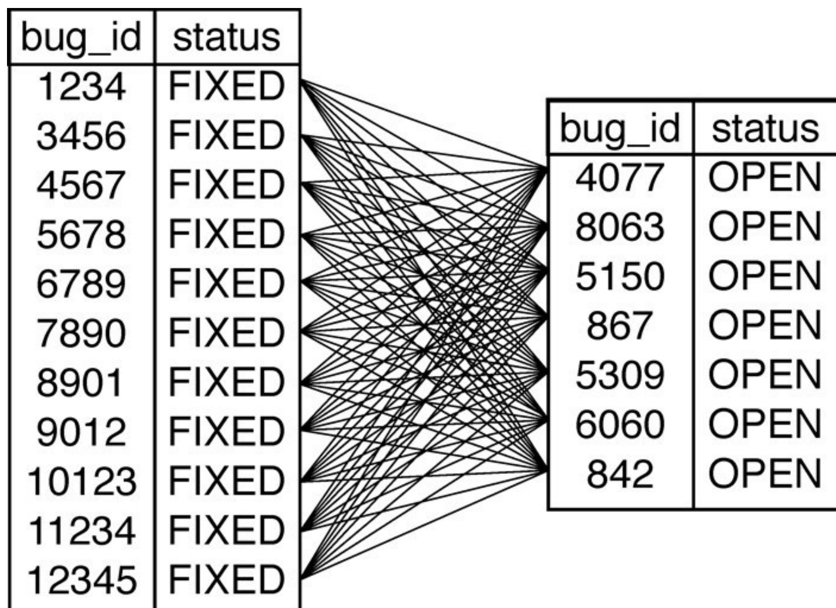


图18-1　被修复与打开Bug的笛卡儿积

Spaghetti-Query/anti/cartesian-no-group.sql

```
SELECT p.product_id, f.bug_id AS fixed, o.bug_id AS open
FROM BugsProducts p
JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status = 'FIXED')
JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status = 'OPEN')
WHERE p.product_id = 1;
```

这个查询唯一描述的关系是BugsProducts表和每个Bugs子集之间的关系。没有条件来约束每个FIXED状态的Bug和每个OPEN状态的Bug是否能进行配对，而默认情况下它们是可以配对的。最终的结果是得到一个12乘以7的结果集。

当你尝试着执行一个类似的双重任务的查询时，很容易就得到一个意料之外的笛卡儿积。如果你尝试在一个查询中完成更多不相关的工作，最终的结果可能是在此基础上再多乘出一个笛卡儿积。

18.2.2　那好像还不够??

除了你会得到错误结果之外，这些查询也非常难写、难以修改和难以调试。数据库查询请求的日益增加应该是预料之中的事。经理们想要更复杂的报告以及在用户界面上添加更多的字段。如果你的设计很复杂，并且是一个单一查询，要扩展它们就会很费时费力。不论对你还是对项目来说，时间花在这些事情上面不值得。

此外，还有运行时开销。一条精心设计的复杂SQL查询，相比于那些直接简单的查询来说，不得不使用很多的JOIN、关联子查询和其他让SQL引擎难以优化和快速执行的操作符。程序员直觉地认为越少的SQL执行次数性能越好，如果SQL查询的复杂度都相同时的确如此。但另一方面，一个怪兽般的SQL查询的开销可能成指数级别增长，而使用多个简单的查询却有更好的效果。

18.3　如何识别反模式

如果你听见项目组成员说了下面这些话，可能就是使用了“意大利面条式查询”这个反模式。

- “为什么我的求和、计数返回的结果异常地大？”

一个意料之外的两个联结查询的数据集生成的笛卡儿积。

- “我一整天都在和这个变态的查询语句做斗争！”

  SQL并不是那么难的，真的！如果你和单条SQL查询纠结了很长时间，应该重新考虑你的实现方式。

- “我们不能在数据库报表中再加入任何新东西了，因为对查询语句的修改要花很长时间。”

  写这个查询语句的人要永远为他所写的代码负责，即使他们已经加入到别的项目中去了。那个写这段代码的人可能就是你，因此别把查询写得如此复杂以至于别人无法维护！

- “试试看再加一个DISTINCT进去？”

  要修正由于笛卡儿积所带来的数据集暴涨，程序员通常使用DISTINCT这个关键字作为一个查询修正或者一个聚合函数来减少重复。这个方法能够隐藏掉那个难看的查询的踪迹，但导致RDBMS做了更多的工作来生成排序、去重的临时表。

另一个表明一个查询可能是意大利面条式查询的证据是它的执行时间很长。低劣的性能可能是其他问题的一个征兆，但你在调查时应该考虑到可能在某个SQL语句中做了太多事情。

18.4 合理使用反模式

需要将一个复杂的查询任务放在一个SQL查询中完成的最常见原因，是你正在使用一个编程框架或者一个可视化组件库直接和数据源相连，然后在程序中直接展示数据。简单的商务智能和报表工具都属于这一分类中，但大多数高级的BI软件可以从多个数据源合并数据。

组件或者报表工具通常假设单个SQL查询仅用来完成一个简单的任务，但它鼓励你去设计更庞大的查询来生成报告中的所有数据。如果你使用某个这样的报表程序，就可能被迫去写一个更复杂的SQL查询，而没有机会写代码操作结果集。

如果报表需求太复杂而不能用单个SQL查询完成，更好的方案可能是生成多个报表。如果你的老板不喜欢这样的解决方案，要提醒他报表的复杂度和生成报表所花的时间是成正比的。

有时候，你想要在一个SQL查询中得到一个复杂的结果，是因为需要所有的这些结果能排序后再组合在一起。在SQL查询中指定一个排序是很简单的。理论上来说让数据库来执行这个操作比你自己写代码实现多个请求结果的排序要高效得多。

18.5　解决方案：分而治之

奥卡姆[1]在本章开头的名句也称为简约律。

> [1] William of Ockham，十四世纪英国著名思想家，逻辑学家。著有“奥卡姆剃刀”理论。——编者注

简约律

> 当你有两个相互竞争的理论能得出同样的结论，那么简单的那个更好。

对于SQL来说，那意味着你在两个能得到同样结果的查询之中做选择时，选择更简单的那个。我们在修正本章的反模式查询时，应时刻谨记这个定律。

18.5.1　一步一个脚印

如果你看不出，在意外产生了笛卡儿积的两张表间存在逻辑关联关系，那有一个很简单的解释：根本就没有这种关系。要避免生成这个笛卡儿积，你不得不将这个意大利面条式查询拆分成几个小而简单的查询。在之前描述的那个简单例子中，我们只需要两个查询：

> Spaghetti-Query/soln/split-query.sql

```
SELECT p.product_id, COUNT(f.bug_id) AS count_fixed
FROM BugsProducts p
LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status =
WHERE p.product_id = 1
GROUP BY p.product_id;

SELECT p.product_id, COUNT(o.bug_id) AS count_open
FROM BugsProducts p
LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status =
```

```
WHERE p.product_id = 1
GROUP BY p.product_id;
```

这两个查询的结果分别是12和7，正如我们所希望的。

你可能会觉得将一个查询拆分成多个查询是一个“不雅”的解决方案，并且略感憋屈和遗憾，但你很快就会意识到，在开发、维护和性能方面，这么做能带来一些积极的影响。

- 这个查询不会产生早先例子中出现的那种意外的笛卡儿积，因此很简单地就能确认这个查询给出的结果是精确的。
- 当有新的需求增加到报表中时，添加另一个简单的查询，比将更多的计算整合到一个已经很复杂的查询中去要简单很多。
- SQL引擎能更容易和可靠地对简单的查询进行优化和执行。即使整个工作看上去像是被分割出来的查询弄得有点重复，但可能执行得更快。
- 在代码审查或者团队间的培训交流时，解释几个易懂的查询要比解释一个庞大复杂的查询容易得多。

18.5.2 寻找UNION标记

你可以将几个查询的结果进行UNION操作，从而最终得到一个结果集。当你确实想要提交单个查询并且得到单个结果集时，这么做很有帮助，比如在需要保存查询结果时。

Spaghetti-Query/soln/union.sql

```
(SELECT p.product_id, f.status, COUNT(f.bug_id) AS bug_count
 FROM BugsProducts p
 LEFT OUTER JOIN Bugs f ON (p.bug_id = f.bug_id AND f.status =
 WHERE p.product_id = 1
 GROUP BY p.product_id, f.status)

UNION ALL

(SELECT p.product_id, o.status, COUNT(o.bug_id) AS bug_count
 FROM BugsProducts p
 LEFT OUTER JOIN Bugs o ON (p.bug_id = o.bug_id AND o.status =
```

```
  WHERE p.product_id = 1
  GROUP BY p.product_id, o.status)

ORDER BY bug_count;
```

这个查询的结果是每个子查询结果联合后所得的。在本例中有两行，每行对应一个子查询。请记住要额外地增加一列来区分不同子查询的结果，本例中是status这一列。

仅在两个子查询的列属性是相互兼容的情况下才能使用UNION。你不能在查询的中间改变列的数值、名字或者数据类型，因此需要确保所有行的所有列都是相同的。如果你发现定义了一个列的别名，类似于bugcount_or_customerid_or_null，很有可能就是你对不兼容的两个结果集使用了UNION。

18.5.3 解决老板的问题

怎么解决这个统计项目信息的紧急任务？你的老板说："我需要知道我们同时在开发多少个项目，多少个程序员在修补Bug，每个程序员平均修补多少个Bug，以及多少修复了的Bug是由用户报告的。"

做好的解决方案就是拆分所有的这些工作。

- 多少产品：

  Spaghetti-Query/soln/count-products.sql

  ```
  SELECT COUNT(*) AS how_many_products
  FROM Products;
  ```

- 多少开发人员在参与修补Bug：

  Spaghetti-Query/soln/count-developers.sql

  ```
  SELECT COUNT(DISTINCT assigned_to) AS how_many_developers
  FROM Bugs
  WHERE status = 'FIXED';
  ```

- 平均每个程序员修复了多少Bug：

> Spaghetti-Query/soln/bugs-per-developer.sql

```
SELECT AVG(bugs_per_developer) AS average_bugs_per_devel
FROM (SELECT dev.account_id, COUNT(*) AS bugs_per_develo
      FROM Bugs b JOIN Accounts dev
        ON (b.assigned_to = dev.account_id)
      WHERE b.status = 'FIXED'
      GROUP BY dev.account_id) t;
```

- 多少修复了的Bug是由客户报告的：

> Spaghetti-Query/soln/bugs-by-customers.sql

```
SELECT COUNT(*) AS how_many_customer_bugs
FROM Bugs b JOIN Accounts cust ON (b.reported_by = cust.
WHERE b.status = 'FIXED' AND cust.email NOT LIKE '%@exam
```

其中的几个查询其本身就已经足够复杂了，要再将它们合并到单个输出结果集中，简直就是噩梦！

18.5.4　使用SQL自动生成SQL

当你拆分一个复杂的SQL查询时，得到的结果可能是很多相似的查询，可能仅仅在数据类型上面有所不同。编写所有的这些查询是很乏味的，因此，最好能够有个程序自动生成这些代码。

代码生成是一种输出一段新的可以编译或者执行的代码的写代码技术。如果手写这些代码很费力，代码生成技术就是非常有价值的。一个代码生成器可以帮你去除重复的工作。

多表更新

> 在做咨询时，我被叫去为另一个部门的经理解决一个紧急的SQL问题。
>
> 我走进了经理办公室，看到他穷途末路的苦恼样子。我们简单地互相问候了一下，他就开始向我解释他所面临的困境：“我希望你

能快速地解决这个问题。我们的库存系统已经下线一整天了。”他并不是SQL的业余开发者，但他告诉我他已经在一个用来同时更新大量记录的SQL语句上花了好几个小时了。

他的问题是他没办法在UPDATE语句上对所有的值使用固定的SQL表达式。事实上，每一行上需要更新的值都是不一样的。他的数据库跟踪一个计算机中心的库存信息以及每台电脑的使用情况。他想要添加一个last_used列记录每台电脑的最后一次使用日期。

他太专注于使用单个SQL语句来解决这个复杂的问题了，这是另一个“意大利面条式查询”的例子！他这几个小时想要写出这个完美的UPDATE的时间，都可以手动更新掉所有的记录了。

Spaghetti-Query/soln/generate-update.sql

```
SELECT CONCAT('UPDATE Inventory '
  ' SET last_used = ''', MAX(u.usage_date), '''',
  ' WHERE inventory_id = ', u.inventory_id, ';') AS upda
FROM ComputerUsage u
GROUP BY u.inventory_id;
```

和他想要写出一个SQL语句来解决这个复杂问题不同，我写了一个脚本来生成一系列更简单且符合需求的SQL语句：

这个查询的输出是一系列的UPDATE语句，由分号分割，可以直接作为SQL脚本执行：

```
Update_statement

UPDATE Inventory SET last_used = '2002-04-19'
WHERE inventory_id = 1234;

UPDATE Inventory SET last_used = '2002-03-12'
WHERE inventory_id = 2345;

UPDATE Inventory SET last_used = '2002-04-30'
WHERE inventory_id = 3456;

UPDATE Inventory SET last_used = '2002-04-04'
WHERE inventory_id = 4567;
```

> 通过这种方法，我在几分钟内就解决了这个经理花了几小时在那里纠结的问题。

执行多次SQL查询或者多条SQL语句可能并不是解决问题最高效的办法，但你应该在效率和解决问题之间找到平衡点。

> 尽管SQL支持用一行代码解决复杂的问题，但也别做不切实际的事情。

第19章　隐式的列

> 连我自己都不知道自己要说什么，怎么告诉你我在想什么？
>
> **E. M.**福斯特

一个PHP开发人员向我寻求帮助，他的图书馆数据库在执行一个看上去很简单的SQL查询时的行为让人疑惑不解。

> Implicit-Columns/intro/join-wildcard.sql

```
SELECT * FROM Books b JOIN Authors a ON (b.author_id = a.auth
```

这个查询返回的所有的书名都是NULL。更奇怪的是，当他执行一个不带Authors表的查询时，返回的结果又是和预期的一样，包含了正确的书名。

我帮他找到了问题的根源：他所使用的PHP数据库扩展将从SQL查询返回的每条记录都表示成一个关联数组。比如，使用$row["isbn"]直接获得Books.isbn的值。在他所设计的表中，Books（书）和Authors（作者）两张表里都有一个title（有“标题”和“称呼”两个意思）列。由于结果数组中的单个条目$row["title"]仅能存储一个值，在这个例子中，Authors.title占据了这个数组条目。而在数据库中，大多数作者的title这一列都没有值，因此$row["title"]的值就等于NULL。当这个查询不包含Accounts表时，列名之间没有冲突，书名这一列就如同我们预期的那样占据了这个数组条目。

我告诉这个程序员，解决方案就是给其中的一个title声明一个别名，这样不同title就会使用数组中不同的条目。

Implicit-Columns/intro/join-alias.sql

```
SELECT b.title, a.title AS salutation
FROM Books b JOIN Authors a ON (b.author_id = a.author_id);
```

随后他问了我第二个问题："我要怎么只给一个列定义别名，同时还要获取其余的所有列？"他想要继续使用通配符（SELECT *），又要给通配符所包含的某一列定义别名。

19.1 目标：减少输入

软件开发人员似乎不太愿意打字，这在某种程度上是对他们选择的这个职业的讽刺，就像欧亨利的小说中那对双胞胎的结局。

程序员通常用下面这个需要写出所有查询列的例子来说明要打的字太多了：

Implicit-Columns/obj/select-explicit.sql

```
SELECT bug_id, date_reported, summary, description, resolutio
  reported_by, assigned_to, verified_by, status, priority, ho
FROM Bugs;
```

程序员喜欢使用SQL通配符，我一点也不觉得惊讶。符号*意味着所有的列，因此列的列表是隐式定义的，而不是显式的。这让查询代码变得更清晰。

Implicit-Columns/obj/select-implicit.sql

```
SELECT * FROM Bugs;
```

同样地，当使用INSERT时，使用默认的方案似乎更好：输入的数据会按照列在表中定义的顺序应用到所有的列上。

Implicit-Columns/obj/insert-explicit.sql

```
INSERT INTO Accounts (account_name, first_name, last_name, em
  password, portrait_image, hourly_rate) VALUES
  ('bkarwin', 'Bill', 'Karwin', 'bill@example.com', SHA2('xyz
```

不需要列出列名让SQL语句变得更短。

Implicit-Columns/obj/insert-implicit.sql

```
INSERT INTO Accounts VALUES (DEFAULT,
  'bkarwin', 'Bill', 'Karwin', 'bill@example.com', SHA2('xyzz
```

19.2 反模式：捷径会让你迷失方向

尽管使用通配符和未命名列能够达到减少输入的目的，但这个习惯也会带来一些危害。

19.2.1 破坏代码重构

假设你要向Bugs表里增加一列，比如说用来安排日程的date_due列。

Implicit-Columns/anti/add-column.sql

```
ALTER TABLE Bugs ADD COLUMN date_due DATE;
```

INSERT语句现在会报错，因为现在这张表需要12个传入参数，而你只有11个。

Implicit-Columns/anti/insert-mismatched.sql

```
INSERT INTO Bugs VALUES (DEFAULT, CURDATE(), 'New bug', 'Test
    NULL, 123, NULL, NULL, DEFAULT, 'Medium', NULL);

-- SQLSTATE 21S01: Column count doesn't match value count at
```

在使用隐式模式执行INSERT时，输入必须严格按照定义表时的那些列的顺序。如果列变了，这条语句就会抛出一个错误，甚至有可能把数据写到错误的列里面去。

假设你要执行一个SELECT *的查询，由于不知道具体的列名，所以你只能按照最开始设计表的顺序来使用结果：

Implicit-Columns/anti/ordinal.php

```
<?php
$stmt = $pdo->query("SELECT * FROM Bugs WHERE bug_id = 1234");
$row = $stmt->fetch();
$hours = $row[10];
```

但是在你不知道的情况下，有人删掉了一列：

Implicit-Columns/anti/drop-column.sql

```
ALTER TABLE Bugs DROP COLUMN verified_by;
```

hours这一列已经不是第十列了，于是程序错误地使用了另一列的数据。在重命名、添加、删除列的时候，程序代码并不能适应查询结果的改变。如果使用了通配符，就无法预测这个查询会返回多少行。

这些错误可能会隐藏得很深，当你在程序的输出中发现问题的时候，很难回溯并定位到出问题的那行代码。

19.2.2 隐藏的开销

在查询中使用通配符可能会影响性能和扩展性。一次查询所获取的列越多，客户端程序和数据库之间的网络传输的字节数也越多。

生产环境中的程序可能会有很多并发的查询请求。它们都共享同一个网络带宽，即使一个千兆网络环境也可能由于上百个客户端同时查询返回上千条记录而造成阻塞。

诸如Active Record这类的对象关系映射（ORM）技术通常默认使用SELECT *取得的数据来填充一个表示数据库行的对象。即使ORM提供了一些修改默认行为方式的接口，大多数程序员也懒得去改。

19.2.3 你请求，你获得

我所遇到的程序员使用SQL通配符时问得最多的问题是："有没有选择

除了几个我不想要的列之外所有列的方法？”可能这些程序员想要避免获取庞大的TEXT类型的列的开销，但他们同时也想要获得通配符所带来的书写上的便利。

答案是“没有”。SQL不支持这种“除了我不想要的，其他都要”的语法。你只能使用通配符获取一张表的所有列，或者一个个显式地列出所有你想要的列。

19.3　如何识别反模式

如下的情形可能预示着你的项目在使用隐式列的时候处理得不恰当，并且造成了一定的麻烦。

- “程序由于还使用老的列名而挂掉了。我们尝试了更新所有相关的代码，但可能还有地方漏掉了。”

  你改变了数据库里的一张表——添加、删除、重命名列，或者改变列的顺序——但没能更新全部使用到这张表的代码。要找到所有对这张表的引用是件工作量很大的事情。

- “我们花了几天时间终于找到了网络的瓶颈，最终我们减小了到数据库服务器的庞大的通信量。根据我们的统计信息，平均每个查询请求获取2MB的数据，但只有十分之一是用来显示的。”

  你获取了一堆用不到的数据。

19.4　合理使用反模式

在你只是为了快速地写几个脚本对一个解决方案进行测试，或者写临时SQL查询对当前数据进行校验时，使用通配符是很合情合理的。只执行一次的查询对可维护性没有任何要求。

本书中的例子用到了通配符，一来是为了节省空间，二来是为了避免分散读者对例子中那些更重要部分的注意力。在实际的工作代码中，我很少使用通配符。

如果你的程序需要在增加、删除、重命名或者重新配置列时依旧能自动适应及调整，那最好还是使用通配符，但要确认对之前描述的那些陷阱有充分的准备。

你可以对一个联结查询中的每个独立的表使用通配符。在通配符之前加上表名或者别名作为前缀。这么做可以让你在从一张表中获取所有列的同时，从另一张表中获取少量你所指定的列。如下例：

Implicit-Columns/legit/wildcard-one-table.sql

```
SELECT b.*, a.first_name, a.email
FROM Bugs b JOIN Accounts a
  ON (b.reported_by = a.account_id);
```

输入一个很长的列的列表是很耗时的。对某些人来说，开发效率比程序执行效率更重要。类似地，你可能会把“写更短更可读的查询语句”的优先级提高。使用通配符确实能减少输入的量，得到一个更简短的查询，因此，如果你确实注重这方面的需求，那就用通配符吧。

我听过一个开发人员抱怨说，从程序传递到SQL查询请求的包太大而使得网络负载加剧，在某些情况下查询语句的长度的确会造成影响。但更常见的情况是，返回的数据所使用的带宽比查询语句本身要多得多。你需要自己判断这些特殊的情况，别纠结于这些小问题。

19.5　解决方案：明确列出列名

每次查询时都列出所有你需要的列，而不是使用通配符或者隐式列的列表。

Implicit-Columns/soln/select-explicit.sql

```
SELECT bug_id, date_reported, summary, description, resolutio
  reported_by, assigned_to, verified_by, status, priority, ho
FROM Bugs;
```

Implicit-Columns/soln/insert-explicit.sql

```
INSERT INTO Accounts (account_name, first_name, last_name, em
  password_hash, portrait_image, hourly_rate)
VALUES ('bkarwin', 'Bill', 'Karwin', 'bill@example.com',
  SHA2('xyzzy'), NULL, 49.95);
```

所有这些输入看上去都是很繁重的工作，但非常值得。

19.5.1　预防错误

还记得poka-yoke[1]吗？你在查询时指明所需要选择的列，这能让SQL查询更好地应付错误以及更早地暴露问题。

> [1] 日本工业领域所使用的一种防差错技术，参考第5章。

- 如果这张表中某一列的位置被移动过，它不会对返回结果中这一列的位置造成影响。
- 如果这张表中新加入一列，它是不会出现在查询结果中的。
- 如果从这张表中删除一列，你的查询会得到一个错误——但是这样挺好，因为你直接就能定位到出错的查询语句，而不是在事后追查问题的起因。

如果指定了列名，在使用`INSERT`语句时也能得到类似的好处。你所定义的插入列的顺序会覆盖原始表的定义，并且插入的值会分配到你想要插入的列里。没有在列表里出现的新加入的那列，会自动获得一个默认值或者直接等于`NULL`。如果你引用了一个已经被删除的列，就会得到一个错误信息，这能更早地发现并解决问题。

这是一个尽早出错原则的例子。

19.5.2　你不需要它

如果必须关心软件的可扩展性和程序的吞吐量，你应该检查一下在网络传输过程中可能造成的浪费。在开发和测试环境中，SQL查询所造成的流量上的问题可以忽略不计，但在生产环境中每秒上千次的SQL查询就会造成严重的问题。

一旦你禁止了SQL通配符，就很自然地有针对性地去除那些你不需要的列，同时也意味着更少的输入。这也能使得网络带宽的使用更加有效率。

> Implicit-Columns/soln/yagni.sql

```
SELECT date_reported, summary, description, resolution, statu
```

```
FROM Bugs;
```

19.5.3 无论如何你都需要放弃使用通配符

你从自动售货机买了一包M&M's，包装袋很有用，它能让你很简单地就把这些糖带回办公桌。一旦你打开了包装袋，就需要将M&M's的每一粒糖都视为独立的个体。它们会滚得到处都是。如果你不小心，一些糖还会掉在桌子下面引来臭虫。但是，你想吃到它们就只有打开包装袋。

在SQL查询中，一旦你想要对某一列进行一些表达式计算，或者使用一个列别名，或者由于性能原因排除某一列，你就打开了通配符这个“包装袋”。你不再享有将所有列的集合当成一个包处理所带来的便利，但能够访问到这个包里面的所有内容。

你不可避免地要在查询中引入列别名、函数，或者从列表中排除某列。如果你从一开始就不使用通配符，那之后要对查询进行修改就会变得更加方便。

> 随便拿，但是拿了就必须吃掉。

第四部分　应用程序开发反模式

本部分内容

- 第20章　明文密码
- 第21章　SQL 注入
- 第22章　伪键洁癖
- 第23章　非礼勿视
- 第24章　外交豁免权
- 第25章　魔豆

第20章　明文密码

> 敌人也了解这套系统。
>
> 香农[1]的格言
>
> [1] Shannon，美国数学家，信息论的创始人。——编者注

你接到一个电话，某个人在登录你提供技术支持的程序时遇到了麻烦。

“我是销售部的Pat Johnson。我忘记了我的密码。你能帮我查一下密码是什么吗？”Pat的声音听起来有点怪，还有点急和不好意思。

“对不起，我不能这么做。”你回答道：“我可以帮你重置密码，然后给你的注册邮箱发一封邮件，你可以根据E-mail里的指示重新设置密码。”

电话那头的男人变得更加不耐烦且不可理喻了。“真荒唐！”他说，“我上一家公司的技术支持就可以直接帮我看一下密码是什么。你难道连你自己的工作都做不好？你要我打电话给你的经理吗？”

你自然想要和你的用户保持良好的关系，因此你执行了一个SQL查询看了下Pat Johnson的账号密码，在电话上告诉了他。

这个人挂了电话后，你向同事说道：“真悬啊。我差点就收到Pat Johnson的投诉了。我希望他不会再抱怨了。”

你的同事很困惑。“他？销售部的Pat Johnson是个女的啊。我觉得你可能把她的密码给了一个骗子。”

20.1 目标：恢复或重置密码

我敢打赌，每个有密码的程序都会碰到用户忘记密码的情况。现今大多数程序都通过E-mail的回馈机制让用户恢复或者重置密码。这个解决方案有一个前提，就是这个用户能访问他在注册服务时留下的邮箱。

20.2 反模式：使用明文存储密码

在这种恢复密码的解决方案中，很常见的一个错误是允许用户申请系统发送一封带有明文密码的邮件。这是数据库设计上一个可怕的漏洞，并且它会导致一系列安全问题，可能会使得未取得授权的人获得系统访问权限。

接下来的几段我们就来分析这些风险。假设我们的错误跟踪系统有一张`Accounts`表，每个用户的账号信息都是这张表里的一条记录。

20.2.1 存储密码

在`Accounts`表中，我们以最典型的字符串形式存储密码：

Passwords/anti/create-table.sql

```
CREATE TABLE Accounts (
  account_id     SERIAL PRIMARY KEY,
  account_name   VARCHAR(20) NOT NULL,
  email          VARCHAR(100) NOT NULL,
  password       VARCHAR(30) NOT NULL
);
```

你可以很简单地通过插入一条带有指定密码的记录来创建一个新账号：

Passwords/anti/insert-plaintext.sql

```
INSERT INTO Accounts (account_id, account_name, email, passwo
  VALUES (123, 'billkarwin', 'bill@example.com', 'xyzzy');
```

使用明文存储密码或者使用明文在网络上传递密码都是不安全的。如果攻击者能够截获你用来插入密码的SQL语句，他们就能直接读到密码。在更新密码或者验证用户是否输入正确的密码时这么做，也会导致同样的问题。黑客有好几种方法能够盗取用户密码，比如下面几种。

- 在客户端和服务器端数据库交互的网络线路上截获数据包。这么做比你想象的要容易得多。有很多这样的软件能做到这一点，比如Wireshark[2]。

  [2] Wireshark（也叫Ethereal），其官网为 http://www.wireshark.org/。

- 在数据库服务器上搜索SQL的查询日志。要这么做的前提是，黑客能够访问到数据库所在的服务器，假设他们真的能登录到服务器上，他们就可以查看那些带有SQL语句的数据库执行日志。

- 从服务器或者备份介质上读取数据库备份文件内的数据。你的备份文件妥善保管了吗？你在回收或者丢弃备份设备之前彻底清理干净里面的数据了吗？

20.2.2　验证密码

一段时间后，当用户想要登录，你的程序需要比较用户输入的密码和数据库中存储的密码是否一致。由于密码本身是以明文存储的，所以这样的比较也是以明文字符串的形式进行的。比如，你可以使用如下的查询来返回0或1，判断用户输入的密码和数据库中的是否一致：

Passwords/anti/auth-plaintext.sql

```
SELECT CASE WHEN password = 'opensesame' THEN 1 ELSE 0 END
  AS password_matches
FROM Accounts
```

```
WHERE account_id = 123;
```

在上面的例子中，用户输入的密码opensesame是错的，因此整个查询返回了0。

就像上一段存储密码中所说的那样，将用户输入的字符串以明文的形式插入到SQL语句中会让攻击者更容易得手。

别把两个不同的候选项绑在一起

> 大多数时候，我所看到的认证查询是将`account_id`和`password`两列同时放在`WHERE`子句里进行匹配查找：
>
> Passwords/anti/auth-lumping.sql
>
> ```
> SELECT * FROM Accounts
> WHERE account_name = 'bill' AND password = 'opensesame';
> ```
>
> 在账号不存在或者用户输入的密码不正确时，整个查询返回空。你的程序无法区分到底认证为什么失败了。最好是使用一个能区分这两种情况的查询方法。那样就可以根据错误合适地选择处理方式了。
>
> 比如，当发现在短时间内同一个账号有很多失败的登录请求时，你可能会想暂时冻结这个账号，因为这可能是一次恶意攻击。然而，所面临的问题是你使用的查询语句没办法区分到底是用户名输错了，还是密码输错了。

20.2.3 在E-mail中发送密码

由于密码在数据库中是以明文形式存储的，你可以很简单地在程序中获取密码：

> Passwords/anti/select-plaintext.sql

```
SELECT account_name, email, password
FROM Accounts
WHERE account_id = 123;
```

随后你的程序就可以根据用户的请求将密码发送到用户的邮箱里。你

可能在访问过网站的密码提醒功能里看到过这样的邮件。比如下面这种类型：

密码恢复邮件实例：

> From: daemon
>
> To: bill@example.com
>
> Subject: password request
>
> 你名为“bill”的账户请求密码提醒。
>
> 你的密码是"xyzzy"。
>
> 点击如下链接登录你的账户：
>
> http://www.example.com/login

将明文密码通过邮件发送是非常严重的安全隐患。E-mail可能会被黑客劫持、记录或者使用多种方式存储。就算使用安全协议查看邮件，收发邮件的服务由值得信赖的系统管理员维护，也不能保证一定安全。由于E-mail的收发都需要经由网络层传输，数据可能会在其他的路由节点上被截获。E-mail安全协议的覆盖面并不足够广，也不是你能控制的。

20.3 如何识别反模式

任何能够恢复你的密码，或者将你的密码通过邮件以明文或可逆转加密的格式发给你的程序，都必然犯了本章的反模式。如果你的程序可以通过一个合法的方式获得用户的明文密码，那么黑客也可以做到同样的事情，只不过是不合法而已。

20.4 合理使用反模式

> 程序开发的道德标准
>
> 如果你在开发一个需要密码的程序，被要求设计一个恢复密码的特性，你应该拒绝这样的需求，提醒项目决策者这么做的风险，然后提供另一个能起到同样作用的解决方案。

> 正如一个电工应该能辨别并改正一个会导致火灾的接线设计一样，作为一个软件工程师，有责任和义务去了解相关的安全问题，并提升软件的安全性。
>
> 你可以阅读关于软件安全方面的*19 Deadly Sins of Software Security*[HLV05]一书。另一个相关资源是Open Web Application Security项目（http://owasp.org）。

你的程序可能需要使用密码来访问一个第三方的服务——这意味着，你的程序可能是一个客户端，必须用可读的格式来存储这个密码。较好的做法是，使用一些程序能够解码的加密方法来存储，而不是直接使用明文的方式存储在数据库中。

你可以将身份认证和验证区分开来。用户可以随意说他自己是谁，但验证的逻辑就是用来证明他确实是他自己。密码就是最常被用来做验证这件事情的。

如果不能确保系统足够安全能抵御有技巧和有针对性的攻击，那么实际上系统只有认证机制而没有可靠的验证机制。但这并不一定是必须的。

并不是所有的程序都有被攻击的风险，也不是所有的程序都有敏感的需要保护的信息。比如说，一个可能只有几个可靠的内部人员访问的内部程序，认证机制就可能足够了。在那些非正式的环境中，一个简单的登录框就已经足够了。额外的建立一个强验证系统可能并不合理。

尽管如此，还是要小心——程序的使用总是有超出它们原来设定环境和作用的趋势。在你的小型内部程序暴露在公司防火墙之外之前，你应该要找个安全专家来评估一下它的安全性。

20.5　解决方案：先哈希，后存储

本章的反模式所描述的主要问题是密码的原始存储格式是可读的。其实可以不将密码读出来就验证用户输入的密码是否正确。这一节就介绍了怎样在SQL数据库中实现这样的安全存储密码的方案。

20.5.1　理解哈希函数

使用单向哈希函数对原始密码进行加密，哈希是指将输入字符串转化

成另一个新的、不可识别的字符串的函数。使用哈希函数后，连原始输入串的长度也变得难以猜测了，因为哈希函数返回的字符串的长度是固定的。比如，SHA-256算法将我们的密码xyzzy，转换成了一个256位的串，若使用16进制表示是一个64字节的字符串。

```
SHA2('xyzzy') =
'184858a00fd7971f810848266ebcecee5e8b69972c5ffaed622f5ee
```

哈希的另一个特征就是不可逆。由于哈希函数的算法设计就是要“丢失”一些输入串的信息，所以你无法从一个哈希串恢复出原始输入串。一个好的哈希算法应该需要花上和直接猜测密码差不多的工作量才能找到原始串。

曾经比较流行的哈希算法是SHA-1，但研究者最近证明，这个只有160位的哈希算法还不够安全，有一种技术能通过哈希串推算出输入串。这项技术非常耗时，但无论如何，都比靠猜破解密码所花的时间短。美国国家标准和技术协会（NIST）宣布，2010年后开始逐步取消SHA-1作为安全哈希算法的资格，取而代之的是其更强大的变异算法[3]：SHA-224、SHA-256、SHA-384和SHA-512。无论是否遵循NIST的标准，至少使用SHA-256算法加密密码总是好的。

> [3] http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html。

MD5是另一个流行的哈希函数，产生128位的哈希串。MD5也被证明是弱加密，因此你最好不要用它来加密密码。稍弱的算法还是有很多使用场景的，但对于诸如密码一类的敏感信息，它们并不适用。

20.5.2　在SQL中使用哈希

下面是对`Accounts`表的重定义。SHA-256的哈希密码总是一个64字节的字符串，因此这一列的类型是固定长度的`CHAR`。

> Passwords/soln/create-table.sql

```
CREATE TABLE Accounts (
  account_id      SERIAL PRIMARY KEY,
  account_name    VARCHAR(20),
  email           VARCHAR(100) NOT NULL,
  password_hash   CHAR(64) NOT NULL
);
```

哈希函数并不是标准的SQL语言，因此你可能要依赖于所使用数据库提供的哈希扩展。比如，MySQL 6.0.5的SSL扩展支持包含了`SHA2()`的函数，它默认返回一个256位的哈希串。

Passwords/soln/insert-hash.sql

```
INSERT INTO Accounts (account_id, account_name, email, passwo
  VALUES (123, 'billkarwin', 'bill@example.com', SHA2('xyzzy'
```

你可以在存储和验证用户输入时使用同样的哈希算法，并对两个哈希后的值进行比较。

Passwords/soln/auth-hash.sql

```
SELECT CASE WHEN password_hash = SHA2('xyzzy') THEN 1 ELSE 0
  AS password_matches
FROM Accounts
WHERE account_id = 123;
```

你可以通过简单地将用户的密码改成一个永远不可能通过哈希函数得到的字符串将其锁住。比如，noaccess这个字符串包含了非十六进制字符，是不可能由哈希函数返回的。

20.5.3 给哈希加料

如果你使用哈希串替代了原始密码，然后攻击者获得了对数据库的访问权限（比如他翻了你的垃圾桶，找到了被丢弃的备份CD），他仍旧可以通过试错法获取用户密码。要猜出每个密码可能会花很长时间，但他可以预先准备好自己的数据库——存储可能的密码和对应的哈希串，然后和从你的数据库中找到的哈希串进行比较。只要有一个用户选择了字典中存在的单词，攻击者就能够很轻易地通过搜索两边的哈希值来找到对应的密码原文。他甚至可以直接使用SQL来做这件事：

Passwords/soln/dictionary-attack.sql

```
CREATE TABLE DictionaryHashes (
  password        VARCHAR(100),
  password_hash   CHAR(64)
);
```

```
SELECT a.account_name, h.password
FROM Accounts AS a JOIN DictionaryHashes AS h
  ON a.password_hash = h.password_hash;
```

防御这种“字典攻击”的一种方法是给你的密码加密表达式加点佐料。具体方法是在将用户密码传入哈希函数进行加密之前，将其和一个无意义的串拼接在一起，即使用户选择了一个在字典中存在的单词作为密码，对加料密码进行哈希得到的串是不太会出现在攻击者的哈希数据库中的，你可以发现增加了随机串得到的哈希值和原始值是不一样的：

```
SHA2('password')
  = '5e884898da28047151d0e56f8dc6292773603d0d6aabbdd62a11ef72

SHA2('password-0xT!sp9')
  = '7256d8d7741f740ee83ba7a9b30e7ac11fcd9dbd7a0147f4cc83c62d
```

每个密码都应该配上不同的随机串，这样攻击者就必须为每个密码都创建一个新的哈希字典。然后他就会回到起点上，因为破解数据库中的密码所花的时间和靠猜达到目的的时间差不多[4]。

> [4] 从哈希值恢复密码的一种更优雅的技术叫做彩虹表，它的性能令人吃惊，但引入随机串也能防御这种技术。

Passwords/soln/salt.sql

```
CREATE TABLE Accounts (
  account_id     SERIAL PRIMARY KEY,
  account_name   VARCHAR(20),
  email          VARCHAR(100) NOT NULL,
  password_hash  CHAR(32) NOT NULL,
  salt           BINARY(8) NOT NULL
);

INSERT INTO Accounts (account_id, account_name, email,
    password_hash, salt)
  VALUES (123, 'billkarwin', 'bill@example.com',
    SHA2('xyzzy' || '-0xT!sp9'), '-0xT!sp9');
```

```
SELECT (password_hash = SHA2('xyzzy' || salt)) AS password_ma
FROM Accounts
WHERE account_id = 123;
```

佐料的合理长度应该是8个字节。你需要为每个密码随机生成佐料。之前的几个例子里，佐料字符串使用的都是可打印字符，但事实上你可以使用随机的、不可打印的任意字符。

20.5.4 在SQL中隐藏密码

现在，你在存储密码之前已经有了一个强哈希函数来对密码进行加密，并且使用了随机字符串来阻止字典攻击，你可能认为这样已经足够安全了。但是，密码还是会在SQL表达式中以明文的形式出现，这意味着如果攻击者截获了网络通信的数据包，或者记录了相关的查询语句的日志给到了错误的人手里，密码就泄露了。

只要不将明文密码放到SQL查询语句中，就能避免这种类型的泄露。你所需要做的是在程序代码中生成密码的哈希串，然后在SQL查询中使用哈希串。这么做的好处是，即使攻击者截获了数据包，他也没办法将哈希反转成他所需要的密码。

即使这么做，你还是需要在哈希之前给原始密码追加随机字符串。

下面是一个PHP的例子，使用了PDO扩展来获得佐料、计算得到哈希串，然后执行查询请求来和数据库中存储的加料哈希值进行比较：

Passwords/soln/auth-salt.php

```
<?php

$password = 'xyzzy';

$stmt = $pdo->query(
   "SELECT salt
    FROM Accounts
    WHERE account_name = 'bill'");

$row = $stmt->fetch();
$salt = $row[0];

$hash = hash('sha256', $password . $salt);
```

```
$stmt = $pdo->query("
  SELECT (password_hash = '$hash') AS password_matches;
  FROM Accounts AS a
  WHERE a.acct_name = 'bill'");
$row = $stmt->fetch();
if ($row === false) {
  // account 'bill' does not exist
} else {
  $password_matches = $row[0];
  if (!$password_matches) {
    // password given was incorrect
  }
}
```

这个hash()函数能确保返回的一定是十六进制数，因此不会有SQL注入的风险（详情参阅第21章）。

在网络程序中，还有另一个地方是攻击者有机会截获网络数据包的：在用户的浏览器和网站服务器之间。当用户提交了一个登录表单时，浏览器将用户的密码以明文形式发送到服务器端，随后服务器端才能使用这个密码进行之前所介绍的哈希运算。你可以通过在用户的浏览器发送表单数据之前就进行哈希运算来解决这个问题。但这个方案也有一些不足的地方，就是你需要在进行正确的哈希运算之前，还要通过别的途径获得和这个密码相关联的佐料。折中方案就是在从浏览器向服务器端提交密码表单时，使用安全的HTTP（https）链接。

20.5.5　重置密码，而非恢复密码

现在，密码已经以一个更安全的方法存储了，但你还是需要解决最原始的问题：帮助那些忘记密码的用户。由于现在数据库中存着的密码是哈希串而不是原始密码，你已经无法恢复他们的密码了。你没办法比一个攻击者更快速地反转一个哈希值，但可以允许用户用别的途径获得访问权限。这里介绍两个简单方案。

第一个方案是当用户忘记他们的密码请求帮助的时候，程序发送一封带有临时生成密码的邮件给用户，而不是直接发给他自己的密码。为了安全起见，在一个较短的时间之内，这个临时密码就会过期，即使这封E-mail被截获了，程序还是不会允许未验证的访问。同时，程序应该设计成一旦用户使用临时密码登录后，就应该被强制要求修改密码。

系统生成临时密码邮件

> From: daemon
>
> To: bill@example.com
>
> Subject: password reset
>
> 你要求重置你账户的密码。
>
> 你的临时密码是"p0trz3ble"。
>
> 一小时之后，这个密码将不能使用。
>
> 点击如下链接登录你的账号并设置你的密码：
>
> http://www.example.com/login

第二个方案，在数据库中记录下这个请求，并且为其分配一个唯一的令牌作为标识，而不是发送带有新密码的邮件：

> Passwords/soln/reset-request.sql

```
CREATE TABLE PasswordResetRequest (
  token        CHAR(32) PRIMARY KEY,
  account_id   BIGINT UNSIGNED NOT NULL,
  expiration   TIMESTAMP NOT NULL,
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id)
);

SET @token = MD5('billkarwin' || CURRENT_TIMESTAMP);

INSERT INTO PasswordResetRequest (token, account_id, expirati
  VALUES (@token, 123, CURRENT_TIMESTAMP + INTERVAL 1 HOUR);
```

随后你可以在E-mail中包含这个令牌。你也可以使用别的途径发送这个令牌，比如短信，只要能将消息送达到这个请求重置密码的账号所有者即可。使用这种方法，如果一个陌生人非法地请求了一次密码重置，系统只会发送E-mail给这个账号的实际拥有者。

密码重置页面临时链接邮件

> `From: daemon`
>
> `To: bill@example.com`
>
> `Subject: password reset`
>
> 你要求重置你账户的密码。
>
> 在一小时内点击下面链接来改变密码。
>
> 一小时之后，这个链接将无法工作，你的密码保持不变。
>
> `http://www.example.com/reset_password?token=f5cabff22532bd0025118905bdea50da`

当程序收到一个从重置密码页面来的请求，令牌的值必须存在于密码重置请求表中，并且该行的过期时间点必须是一个将来的时间点而不是过去的时间点。同时，该行的`account_id`引用`Accounts`表，因此，这个令牌被约束为只能重置一个指定的账号。

当然，如果其他人访问了这个页面，也会造成问题。可以通过一些简单的方法来减小风险，比如这个特殊页面的有效期非常短，并且这个页面上不会显示哪个账号的密码要被重置等。

密码学在不断进步，努力保持领先于攻击技术。本章所介绍的这些技术能够帮助改进很多一般的程序，但如果你所开发的系统对安全性要求非常高，你应该深入研究一些更高级的技术，比如下面提到的。

- PBKDF2（http://tools.ietf.org/html/rfc2898）是一个被广泛使用的密码加强标准。
- Bcrypt（http://bcrypt.sourceforge.net/）实现了一个自适应哈希函数。

> 如果密码对你可读，那对攻击者也如此。

第21章　SQL注入

> 就像我说的，我被错误地引述了。
>
> 格劳乔·马克思

2010年3月，美国历史上最大规模的身份盗窃案告破，黑客Albert Gonzalez伏法。他通过侵入ATM机、几家大型零售商的系统以及相关联的信用卡公司的系统，总计盗窃了1.3亿个信用卡与借记卡信息。

Gonzalez打破了他自己保持的盗窃记录，原来的记录是他使用了无线网络的漏洞，在2006年盗窃了4560万个信用卡与借记卡信息。

Gonzalez是怎么能做到的？我们可以想象一下邦德电影里的场景，从电梯井放下绳索，使用超级计算机，破解最先进的加密算法，或者破坏整座城市的电力系统之类的。

起诉书中对案情的描写要真实平常得多。Gonzalez利用了互联网上最常见的安全漏洞。他使用了SQL注入的攻击手段，获得了往受害服务器上传文件的权限，随后他和同伙获得了对系统的访问权限。起诉书[1]中有如下这样的描述。

> [1] http://voices.washingtonpost.com/securityfix/heartlandIndictment.pdf。

攻击执行手段：恶意软件

> 他们会在受害者的电脑上安装一个叫做sniffer的程序，这个程序会在受害人使用网络进行交易时，收集他的信用卡账号和相关信息，然后定期地将这些数据发送给这伙人。

被Gonzalez攻击的那几家零售商声称，他们已经修复了这些安全漏洞。然而，他们堵住了一个漏洞，每天都有新的带有其他漏洞的网络程序被发布。SQL注入对于黑客来说仍然是一个很容易的突破口，因为软件开发人员并不了解这个漏洞的原理，以及该如何防止这样的攻击。

21.1　目标：编写SQL动态查询

SQL常和程序代码一起使用。我们通常所说的**SQL动态查询**[2]，是指将程序中的变量和基本SQL语句拼接成一个完整的查询语句。

> [2] 技术上来说，任何在运行时解析的查询都是SQL动态查询，但通常在实际中，SQL动态查询指的是在语句中包含变量数据的查

询请求。

SQL-Injection/obj/dynamic-sql.php

```
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id = $bug_id";
$stmt = $pdo->query($sql);
```

这个简单的例子展示了如何将PHP变量插入到一个SQL查询语句中。我们期望$bug_id是一个整型，因此当数据库接收到这个请求时，$bug_id的值就是查询语句的一部分。

SQL动态查询是有效利用数据库的很自然的方法。当你使用程序内的变量来指定如何进行查询时，就是在将SQL作为链接程序和数据库的桥梁。程序和数据库之间通过这种方式进行“对话”。

然而，要让程序按照你想要的方式执行并不难，难的是要让程序变得安全，不执行你不想让它执行的操作。但软件在受到SQL注入攻击时，通常都无法保证安全。

21.2 反模式：将未经验证的输入作为代码执行

当往SQL查询的字符串中插入别的内容，而这些被插入的内容以你不希望的方式修改了查询语句的语法时，SQL注入就成功了。在传统的SQL注入案例中，所插入的内容首先完成一个查询，然后再执行第二个完整的查询逻辑。比如，如果$bug_id的值是1234; DELETE FROM Bugs，之前例子中的查询语句最终会变成这样：

SQL-Injection/anti/delete.sql

```
SELECT * FROM Bugs WHERE bug_id = 1234; DELETE FROM Bugs
```

这种类型的SQL注入比较明显，就如图21-1[3]的这个故事一样。通常这些缺陷隐藏得比较好，但还是会有危险。

3 漫画由Randall Munroe所绘（http://xkcd.com/327）已经许可。

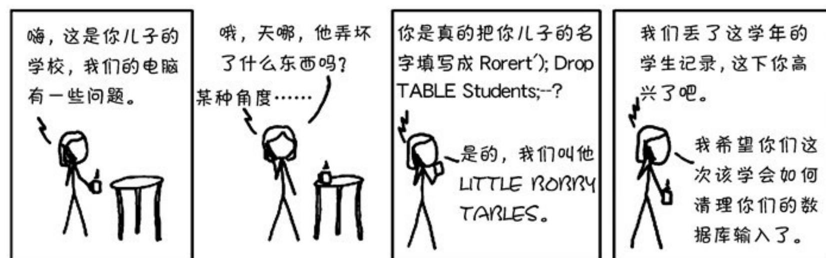


图21-1　妈妈的功绩

21.2.1　意外无处不在

假设你正在为Bug数据库编写一个Web程序，其中的一个页面允许你根据名字访问项目：

SQL-Injection/anti/ohare.php

```
<?php
$project_name = $_REQUEST["name"];
$sql = "SELECT * FROM Projects WHERE project_name = '$project_
```

当你的开发小组开始为芝加哥的O’Hare国际机场开发软件时，问题出现了。你们很自然地给这个项目取名为“O’Hare”，在你们的Web程序中，要怎样提交一个请求才能看到这个项目呢？

```
http://bugs.example.com/project/view.php?name=O'Hare
```

你的PHP代码接受了请求参数并插入到SQL查询语句中，但是实际生成的SQL语句却不是大家所期望的：

SQL-Injection/anti/ohare.sql

```
SELECT * FROM Projects WHERE project_name = 'O'Hare'
```

由于字符串是被两个单引号所包围，生成的SQL语句中包含了一个很

短的'O'，然后是一些额外的字符，在这个例子中，Hare'没有任何意义。对于这样的SQL语句，数据库只能返回一个语法错误的提示。这的确是一个意外。发生任何不好的事情的风险还是比较小的，因为语法错误的SQL语句是不会被执行的。风险较大的情况是产生的SQL没有任何语法错误，并且以一种你所不希望的方式执行。

21.2.2　对Web安全的严重威胁

当攻击者能够使用SQL注入操控你的SQL查询语句时，它就变成了一个巨大的威胁。比如，你的程序可能允许用户以如下方式修改密码：

SQL-Injection/anti/set-password.php

```
<?php
$password = $_REQUEST["password"];
$userid = $_REQUEST["userid"];
$sql = "UPDATE Accounts SET password_hash = SHA2('$password')
    WHERE account_id = $userid";
```

一个聪明的攻击者会猜测你的每个请求参数对应在SQL语句中的作用，并且精心选择每个参数对应的值：

```
http://bugs.example.com/setpass?password=xyzzy&userid=123 OR
```

通过在userid这个参数后插入额外的字符串，攻击者改变了对应的SQL语句的意义。现在，这条语句会改变每一个用户的密码，而不是只改变一个用户：

SQL-Injection/anti/set-password.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = 123 OR TRUE;
```

这是理解SQL注入的关键，也是如何防止SQL注入的关键：SQL注入是通过在SQL语句被数据库解析之前，以修改其语法的形式工作的。只要你在解析语句之前插入动态部分，就存在SQL注入的风险。

有数不尽的方法选择一个恶意的字符串来改变SQL语句的行为。它只受制于攻击者的想象力和你保护SQL语句的能力。

21.2.3 寻找治愈良方

既然我们已经知道了SQL注入的风险，下一个问题便是：要做什么来使代码远离SQL注入呢？可能你之前读过一些博文或者一些文章，声称某一种技术是对抗SQL注入的万能药。而事实上，这些技术都被证明无法阻挡所有类型的SQL注入。因此你需要在不同的情况下将所有这些技术组合起来使用。

转义

防止SQL语句包含任何不匹配的引号的最古老的方法，就是对所有的引号字符进行转义操作，使它们不至于成为字符串的结束符。在标准SQL中，可以使用两个连续的单引号来表示一个单引号字符：

> SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Projects WHERE project_name = 'O''Hare'
```

大多数的数据库还支持使用反斜杠对单引号字符进行转义操作，就和大多数其他的编程语言一样：

> SQL-Injection/anti/ohare-escape.sql

```
SELECT * FROM Projects WHERE project_name = 'O\'Hare'
```

这么做的原理是，在将应用程序中的数据插入到SQL语句之前就进行转换。大多数SQL的编程接口都会提供一个简便的函数来做这个操作。比如，在PHP的PDO扩展中，可以使用一个`quote()`函数来定义一个包含引号的字符串或者还原一个字符串中的引号字符。

> SQL-Injection/anti/ohare-escape.php

```
<?php
$project_name = $pdo->quote($_REQUEST["name"]);
$sql = "SELECT * FROM Projects WHERE project_name = $project_
```

这种技术能够减少由于动态内容中不匹配的引号所造成的SQL注入的风险。但在非字符串内容的情况下，这种技术就会失效。

SQL-Injection/anti/set-password-escape.php

```
<?php
$password = $pdo->quote($_REQUEST["password"]);
$userid = $pdo->quote($_REQUEST["userid"]);
$sql = "UPDATE Accounts SET password_hash = SHA2($password)
    WHERE account_id = $userid";
```

SQL-Injection/anti/set-password-escape.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = '123 OR TRUE'
```

你没办法让一个数值列直接和一个带有数字的字符串进行比较，不管使用哪款数据库，这都是不可能的。某些数据库可能会隐式地将字符串转换成一个等价的数字，但在标准SQL中，在将字符串转换成数字时，一定要明确使用`CAST()`函数。

在一些案例中，有些使用非ASCII编码的字符串在经过了对引号字符进行转义的函数的处理后，依旧保持引号字符没有任何改变。[4]

4 可以参考 http://bugs.mysql.com/8378。

查询参数

一个经常被认为是防止SQL注入的万能型解决方案是使用查询参数。不同于在SQL语句中插入动态内容，查询参数的做法是在准备查询语句的时候，在对应参数的地方使用参数占位符。随后，在执行这个预先准备好的查询时提供一个参数。

SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Projects WHERE project_n
$params = array($_REQUEST["name"]);
$stmt->execute($params);
```

大多数开发人员都推荐这个方案，因为你不需要对动态内容进行转义，或者担心有缺陷的转义函数。事实上，查询参数这个方法的确是

应对SQL注入的强劲解决方案。但是这还不是一个通用的解决方案，因为查询参数总被视为是一个字面值。

- 多个值的列表不可以当成单一参数：

  SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs WHERE bug_id I
$stmt->execute(array("1234,3456,5678"));
```

  这样的做法会导致数据库认为传入的是一个包含数字和逗号的字符串，处理过程和将一系列整数作为参数进行查询并不一样：

  SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs WHERE bug_id IN ( '1234,3456,5678')
```

- 表名无法作为参数：

  SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM ? WHERE bug_id = 12
$stmt->execute(array("Bugs"));
```

  这么做是想将一个字符串插入表名所在的位置，但只会得到一个语法错误的提示：

  SQL-Injection/anti/parameter.sql

```
SELECT * FROM 'Bugs' WHERE bug_id = 1234
```

- 列名无法作为参数：

  SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY ?");
$stmt->execute(array("date_reported"));
```

在这个例子中，排序是一个无效操作，因为排序表达式是一个常量字符串，对所有行都是一样的：

SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs ORDER BY 'date_reported';
```

- SQL关键字不能作为参数：

SQL-Injection/anti/parameter.php

```
<?php
$stmt = $pdo->prepare("SELECT * FROM Bugs ORDER BY date_
$stmt->execute(array("DESC"));
```

参数将被当做字面字符串插入而非SQL关键字。在这个例子中，会返回语法错误的提示。

SQL-Injection/anti/parameter.sql

```
SELECT * FROM Bugs ORDER BY date_reported 'DESC'
```

> 我的查询是如何完成的
>
> 很多人认为SQL的查询参数就是一种能自动将参数包含进SQL语句的技术。这并不准确，如果仅仅这么想，会导致对这项技术的严重误解。
>
> RDBMS服务器首先会解析你所准备好的SQL查询语句。在完成这一步操作之后，就没有任何方法能够改变那句SQL查询语句的结构。
>
> 在执行一个已经准备好的SQL查询时，你需要提供对应的参数，

> 每个你提供的参数都对应于预先准备好的查询中的一个占位符。
>
> 你可以重复执行这个预先准备好的查询，只要使用新的参数替换掉老的参数就行了。因此，RDBMS系统必须将查询操作和对应的传入参数分开跟踪。这对于系统安全来说是一件好事。
>
> 这意味着，如果你获取到一个预先准备好的SQL查询语句，它里面是不会包含任何实际的参数值的。当你调试或者记录查询时，很方便就能看到带有参数值的SQL语句，但这些值永远不会以可读的SQL形式整合到查询中去。
>
> 调试动态化SQL语句的最好方法，就是将准备阶段的带有占位符的查询语句和执行阶段传入的参数都记录下来。

存储过程

存储过程，是很多程序员声称可以抵御SQL注入攻击的方法。通常来说，存储过程包含固定的SQL语句，这些语句是在定义这个存储过程的时候被解析的。

然而，在存储过程中也是可以使用SQL动态查询的，而这么做也会有安全隐患。在下面的这个例子中，input_userid参数会被直接替换到SQL查询中，而这么做很不安全。

> SQL-Injection/anti/procedure.sql

```
CREATE PROCEDURE UpdatePassword(input_password VARCHAR(20),
  input_userid VARCHAR(20))
BEGIN
  SET @sql = CONCAT('UPDATE Accounts
    SET password_hash = SHA2(', QUOTE(input_password), ')
    WHERE account_id = ', input_userid);
  PREPARE stmt FROM @sql;
  EXECUTE stmt;
END
```

在存储过程中使用SQL动态查询，一点也不比在程序代码中直接使用SQL动态查询安全。这个input_userid参数可以包含有害的内容，并且造成最终执行的SQL语句变得不安全。

> SQL-Injection/anti/set-password.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = 123 OR TRUE;
```

数据访问框架

你可能看过数据访问框架的拥护者声称他们的库能够抵御所有SQL注入的攻击。对于所有允许你使用字符串方式传入SQL语句的框架来说，这都是扯淡。

良好的规范

> 在我做了一个关于我所开发的基于PHP的数据访问框架的讲座之后，一位听众站起来问我："你的框架能抵挡SQL注入吗？"我告诉他，这个框架提供了一些处理字符串的函数和使用查询参数的方法。
>
> 这个年轻人看上去很疑惑。"但是，它能抵挡SQL注入吗?"他重复道。他在寻求一种自动化的方法来确保他没有犯任何自己都无从辨认的错误。
>
> 我告诉他使用框架来抵挡SQL注入，就好比使用牙刷来防止蛀牙。你需要持续使用才能有所帮助。

没有任何框架能强制你写出安全的SQL代码。一个框架可能会提供一系列简单的函数来帮助你，但很容易就能绕开这些函数，然后使用通常的修改字符串的办法来编写不安全的SQL语句。

21.3 如何识别反模式

事实上几乎所有的数据库应用程序都动态地构建SQL语句。如果你使用拼接字符串的形式或者将变量插入到字符串中的方法来构建哪怕一句SQL语句，那这一句查询语句就会让应用程序暴露在 SQL注入攻击的威胁下。

SQL注入攻击非常常见，因而你应该假设在程序中使用了SQL的部分总是存在那么一两个漏洞的。除非你的团队专门针对SQL注入做过一次代码审查，找到并且修复了相关的漏洞。

> 规则**31**：检查车后座

如果你喜欢看怪物电影，就会知道那些可怕的生物总是喜欢藏在车后座，等着你坐进车里。这教导我们，千万别假设你所熟悉的地方就一定安全，比如你的车。

SQL注入可以采用间接的形式。即使你最开始使用查询参数安全地插入用户提供的数据，你也可能在后续的查询中动态地使用这些数据：

```
<?php
$sql1 = "SELECT last_name FROM Accounts WHERE account_id
$row = $pdo->query($sql1)->fetch();
$sql2 = "SELECT * FROM Bugs WHERE MATCH(description) AGA
  . $row["last_name"] .  "')";
```

在上一个查询中，如果用户使用了“O'Hara”这个用户名，或者故意包含了SQL语法的词，又会发生什么呢？

21.4 合理使用反模式

这个反模式和本书中其他的反模式不同，因为没有任何合理的理由允许SQL注入让程序存在安全漏洞。编写能够防御SQL注入的代码，以及帮助你的同事也这么做，是你作为一个软件开发人员的责任。软件的安全性是用其最薄弱的环节来衡量的——确保这个最薄弱的环节不是由你造成的！

21.5 解决方案：不信任任何人

没有哪一种技术能使SQL代码变得安全。你应该学习下面所描述的所有技术，并在合理的地方使用它们。

21.5.1 过滤输入内容

你应该将所有不合法的字符从用户输入中剔除掉，而不是纠结于是否有些输入包含了有危险的内容。也就是说，如果你需要一个整数，那就只使用输入中的整数部分。根据你所使用的开发语言不同，方法也不尽相同，比如在PHP中，可以使用`filter`扩展：

SQL-Injection/soln/filter.php

```
<?php
$bugid = filter_input(INPUT_GET, "bugid", FILTER_SANITIZE_NUM
$sql = "SELECT * FROM Bugs WHERE bug_id = {$bugid}";
$stmt = $pdo->query($sql);
```

对于简单的数字转换，你可以直接使用类型转换函数：

> SQL-Injection/soln/casting.php

```
<?php
$bugid = intval($_GET["bugid"]);
$sql = "SELECT * FROM Bugs WHERE bug_id = {$bugid}";
$stmt = $pdo->query($sql);
```

你还可以使用正则表达式来匹配安全的子串，过滤非法内容：

> SQL-Injection/soln/regexp.php

```
<?php
$sortorder = "date_reported"; // default

if (preg_match("/[_[:alnum:]]+/", $_GET["order"], $matches))
  $sortorder = $matches[1];
}

$sql = "SELECT * FROM Bugs ORDER BY {$sortorder}";
$stmt = $pdo->query($sql);
```

21.5.2 参数化动态内容

如果查询中的变化部分是一些简单的类型，你应该使用查询参数将其和SQL表达式分离。

> SQL-Injection/soln/parameter.php

```
<?php
$sql = "UPDATE Accounts SET password_hash = SHA2(?) WHERE acc
$stmt = $pdo->prepare($sql);
$params = array($_REQUEST["password"], $_REQUEST["userid"]);
$stmt->execute($params);
```

我们看到21.2节中，一个参数只能被替换为一个值。如果你是在RDMBS解析完SQL语句之后才插入这个参数值，没有哪种SQL注入的攻击能够改变一个参数化了的查询的语法结构。即使攻击者尝试使用带有恶意的参数值，诸如123 OR TRUE，RDBMS会将这个字符串当成一个完整的值插入。最坏情况下，这个查询没办法返回任何记录，它不会返回错误的行。这个恶意串最终可能会导致程序生成如下的查询语句，但这条语句无害：

SQL-Injection/soln/parameter.sql

```
UPDATE Accounts SET password_hash = SHA2('xyzzy')
WHERE account_id = '123 OR TRUE'
```

当需要将程序中的变量以字符串形式插入SQL表达式中时，应该使用查询参数。

21.5.3　给动态输入的值加引号

查询参数通常来说是最好的解决方案，但在有些很特殊的情况下，参数的占位符会导致查询优化器无法正确选择使用哪个索引来进行优化。

比如说，假设在Accounts表中有一个is_active列。这一列中99%的记录都是真实值。对is_active = false的查询会得益于这一列上的索引，但对于 is_active = true的查询却会在读取索引的过程中浪费很多时间。然而，如果你用了一个参数is_active = ?来构造这个表达式，优化器不知道在预处理这条语句的时候你最终会传入哪个值，因此很有可能就选择了错误的优化方案。

要规避这样的问题，直接将变量内容插入到SQL语句中会是更好的方法，不要去理会查询参数。一旦你决定这么做了，就一定要小心地引用字符串。

SQL-Injection/soln/interpolate.php

```
<?php
$quoted_active = $pdo->quote($_REQUEST["active"]);
$sql = "SELECT * FROM Accounts WHERE is_active = {$quoted_act
$stmt = $pdo->query($sql);
```

你需要确信所使用的函数是经过严格测试的、不会带有SQL安全隐患的。大多数的数据库访问库都包含一个这样的字符串引用处理函数。比如在PHP中，可以使用`PDO::quote()`。别总想着自己实现一个这样的函数，除非你对所有的安全风险有深入的了解。

21.5.4　将用户与代码隔离

查询参数和转义字符能帮助你将字符串类型的值插入到SQL表达式中，但这些技术在需要插入表/列名或者SQL关键字的时候不起作用。你需要另一项技术来使得这些部分也能动态化。

假设你的用户想要能够选择以什么方式给Bug排序，比如按照状态或者按照提交的日期以及排序的方向。

> SQL-Injection/soln/orderby.sql

```
SELECT * FROM Bugs ORDER BY status ASC
SELECT * FROM Bugs ORDER BY date_reported DESC
```

在如下的例子中，PHP脚本接收`order`和`dir`两个参数，然后代码将用户的选择作为列名和关键字，插入到SQL查询中。

> SQL-Injection/soln/mapping.php

```
<?php
$sortorder = $_REQUEST["order"];
$direction = $_REQUEST["dir"];
$sql = "SELECT * FROM Bugs ORDER BY $sortorder $direction";
$stmt = $pdo->query($sql);
```

这个脚本假设`order`字段包含了一个列名，`dir`字段为ASC或者DESC。但这并不是一个安全的假设，因为用户在网络请求中可以随意传递参数的值。

> 参数化**IN()**操作
>
> 我们知道你不能将一个由逗号分割的字符串当做单个参数传入，你需要跟你的对象列表一样多的参数。
>
> 比如需要通过主键来查询6个Bug，你将这个列表存在数组变量

$bug_list中：

```
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id IN (?, ?, ?, ?,
$stmt = $pdo->prepare($sql);
$stmt->execute($bug_list);
```

仅当$bug_list中正好包含6个值，和占位符的数量完全匹配，上面这条语句才能正常工作。你应该动态地创建IN()这个语句，使用和$bug_list中数据个数一样多的占位符。

下面的PHP例子使用了一些PHP内置的数组函数来生成一个占位符数组，其条目个数和$bug_list的条目个数一致，然后在插入到SQL表达式之前，将这些数组用逗号拼接在一起。

```
<?php
$sql = "SELECT * FROM Bugs WHERE bug_id IN ("
    . join(",", array_fill(0, count($bug_list), "?")) .
$stmt = $pdo->prepare($sql);
$stmt->execute($bug_list);
```

用这个技巧来参数化一个列表。

对应的解决方案是，将请求的参数作为索引值去查找预先定义好的值，然后用这些预先定义好的值来组织SQL查询语句。

1. 声明一个$sortorders数组，将用户的可选项作为索引值，将SQL的列名作为对应的值。声明一个$directions数组，将用户的可选项作为索引，将SQL关键字ASC和DESC作为对应的值。

   SQL-Injection/soln/mapping.php

```
$sortorders = array( "status" => "status", "date" => "da
$directions = array( "up" => "ASC", "down" => "DESC" );
```

2. 当用户的选择不在这两个数组中时，让变量$sortorder和$dir等于默认值。

SQL-Injection/soln/mapping.php

```
$sortorder = "bug_id";
$direction = "ASC";
```

3. 如果用户的选择在`$sortorders`和`$directions`数组中，就使用对应的值。

SQL-Injection/soln/mapping.php

```
if (array_key_exists($_REQUEST["order"], $sortorders)) {
  $sortorder = $sortorders[ $_REQUEST["order"] ];
}


if (array_key_exists($_REQUEST["dir"], $directions)) {
  $direction = $directions[ $_REQUEST["dir"] ];
}
```

4. 现在使用`$sortorder`和`$direction`这两个变量就是安全的了，因为它们只能是在代码中预先定义的值。

SQL-Injection/soln/mapping.php

```
$sql = "SELECT * FROM Bugs ORDER BY {$sortorder} {$direc
$stmt = $pdo->query($sql);
```

这种技术有如下几个优势。

- 从不将用户的输入与SQL查询语句连接，因此减少了SQL注入的风险。
- 可以让SQL语句中的任意部分变得动态化，包括标识、SQL关键字，甚至整句表达式。
- 使用这个方法验证用户的输入变得很简单且高效。
- 能将数据库查询的细节和用户界面解耦。

选项需要硬编码在程序中，但对于表明、列名和SQL关键字来说，这么做还是很恰当的。对于数据值的选择往往是全量的字符串或者数字，但对于语法标识来说，可选择的范围很小。

21.5.5　找个可靠的人来帮你审查代码

找到瑕疵的最好方法就是再找一双眼睛一起来盯着看。你需要找个熟悉SQL注入的同事来帮助你一起检查代码。别让自大和骄傲阻止你做正确的事——现在承认你的代码有问题可能会 让你感到窘迫，但你确定更愿意为黑客利用安全漏洞攻击网站而承担后果吗？

在检查代码是否包含SQL注入风险的时候，参考下面这几点。

1. 找出所有使用了程序变量、字符串链接或者替换等方法组成的SQL语句。

2. 跟踪在SQL语句中使用的动态内容的来源。找出所有的外部的输入，比如用户输入、文件、系统环境、网络服务、第三方代码，甚至于从数据库中获取的字符串。

3. 假设任何外部内容都是潜在的威胁。对于不受信任的内容都要进行过滤、验证或者使用数组映射的方式来处理。

4. 在将外部数据合并到SQL语句时，使用查询参数，或者用稳健的转义函数预先处理。

5. 别忘了在存储过程的代码以及任何其他使用SQL动态查询语句的地方做同样的检查。

代码检查是找出SQL注入缺陷的最正确和经济的方法。你应该预留出相应的时间来做这件事，并且将其视为项目开发过程中一个必要的过程。当然，你可以帮助同事检查代码来回报他们为你所做的。

> 让用户输入内容，但永远别让用户输入代码。

第22章　伪键洁癖

> 重要的人不关心，关心的人不重要。
>
> 伯纳德·巴鲁克（在安排晚会餐桌席次时所说）

你的老板带着两份打印出来的报告过来找你，说：“会计部的人说我们给出的这一季度报告和上季度报告有些差异。我正在看这两份报告，的确有差异，大部分最新的资产消失了。怎么回事？”

你看着这两份报告，发现这些差异看起来很眼熟。“不，每样东西都在那里。为了使所有的记录编号都是连续的，你让我整理过一次数据库。你说会计们由于数字之间的断档，一直在追问你中间那些不见了的资产是怎么回事。

“因此，我重新为一些记录编了号，然后把它们放在了原来的空行。现在没有断档了——从1到12340之间的每个数字都对应一个资产。所有的东西都在那里，只是有些改变了编号并且移到上面去了。是你告诉我这么做的。”

老板不住地摇头。“但这不是我想要的。会计人员是根据资产编号来跟踪设备的折旧状况的。每个设备的编号要在每个季度的报告中保持一致。除此之外，所有的资产编号都被打印并且贴在了对应的设备上。要花好几周的时间来重新为整个公司的设备贴新的标签。你能把所有的ID编号改回原来的吗？”

你想表现得自己很合作，因此转回电脑前开始工作，但你突然想到了一个新问题。“在我将这些资产ID改回原来的之后，这个月买的那些新资产怎么办？有些新资产使用了我重新编号之前就已在使用的编号。如果我把资产ID都改回原来的值，要怎么处理这些冲突？”

22.1 目标：整理数据

确实有这么一种人，他们会为一系列数据的断档而抓狂。

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider

一方面，bug_id 3这一行确实发生过一些事情。为什么这个查询不返回这个Bug？这条记录丢失了吗？那个Bug到底是什么？这个Bug是我们的一个重要客户提交的吗？我要为这条数据的丢失负责吗？

对于具有伪键洁癖这个反模式的那些人来说，他们的目的就是要解决这些麻烦的问题。这些人对数据的完整性问题负责，但通常来说，他们对数据库不够了解，或者不怎么相信数据库所生成的报表。

22.2 反模式：填充角落

大多数人对于断档的第一反应就是想要填补其中的空缺。对此，可能会有两种做法。

22.2.1 不按照顺序分配编号

你可能想要在插入新行时，通过遍历表，将找到的第一个未分配的主键编号分配给新行，来代替原来自动分配的伪主键机制。随着你不断地插入新行，断档就被填补起来了。

bug_id	status	product_name
1	OPEN	Open RoundFile
2	FIXED	ReConsider
4	OPEN	ReConsider
3	NEW	Visual TurboBuilder

然而，为了找到第一个未被使用的值，你不得不执行一个不必要的自联合查询：

Neat-Freak/anti/lowest-value.sql

```
SELECT b1.bug_id + 1
FROM Bugs b1
LEFT OUTER JOIN Bugs AS b2 ON (b1.bug_id + 1 = b2.bug_id)
WHERE b2.bug_id IS NULL
ORDER BY b1.bug_id LIMIT 1;
```

在本书的前几章中，我们解释过在创建唯一标识的主键时，如果使用`SELECT MAX(bug_id)+1`这种查询语句，则会出现并发访问的问题。如果有两个程序同时想要找到最小的未使用值时，也会出现同样的问题，结果就是一个成功，另一个失败。况且，这个方法既低效，也容易导致问题。

22.2.2 为现有行重新编号

在某些情况下，你可能急着想要让所有的主键变得连续，而等着新行来填补空缺不够快。你可能会考虑更新现有行的主键值，让其变得连续，从而消除断档。通常这样的做法是找到主键最大的行，然后用最小的未被使用的值来更新它。比如，你可以将4更新为3：

Neat-Freak/anti/renumber.sql

```
UPDATE Bugs SET bug_id = 3 WHERE bug_id = 4;
```

bug_id	status	product_name
1	NEW	Open RoundFile
2	FIXED	ReConsider
3	DUPLICATE	ReConsider

要完成这一步，你需要使用和前面一种方法中插入新行类似的方法。先找到一个未被使用的值，随后执行UPDATE语句来重新分配主键值。这两步都会引起并发访问的问题。而且，你需要重复执行很多次这样的操作，才能将较大的断档填满。

你必须同时更新所有引用了你重新分配了主键的行的子记录。如果你在定义外键时，使用了ON UPDATE CASCADE选项，这一步会变得很方便，但如果你没这么做，就必须先禁用约束，手动更新所有的子记录，然后恢复约束。这是一个费时费力且容易出错的操作，并且可能会影响到数据库的服务，正常人都会避免这么做的。

即使你完成了清理操作，“整洁”也是个“短命鬼”。当数据库生成一个新的伪键时，这个值是根据上次生成的值来计算的（即使上次生成的这条记录已经被删除了，也不会有任何影响），并不是按照现有记录中的最大值来计算的，这和一些数据库开发人员的假设相违背。假设你将最大的bug_id 4更新为最小的未被使用的值，然后消除了一个断档，下一次你使用默认的伪键生成器插入新行时，生成的伪键是5，这就会在4的地方产生一个新的断档。

22.2.3　制造数据差异

Mitch Ratcliffe说：“计算机是人类历史中最容易让你犯更多错误的发明……除了手枪和龙舌兰之外。”[1]

[1] *MIT Technology Review*，1992年4月。

本章最开始的故事描述了重编号主键所存在的一些隐患。如果别的外部系统依赖于数据库中的主键来定义数据，那么你的更新操作就会使那个系统中的引用失效。

重用主键并不是一个好主意，因为断档往往是由于一些合理的删除或者回滚数据所造成的。比如，假设一个`account_id`为789的用户由于发送带有人身攻击性的邮件而被封号。产品策略要求你删除这个账号，但如果你重用了主键，就会在某一个时间点将789分配给另一个用户。由于收件人可能在删除后的某一个时间点才打开这些邮件，他们还会投诉789这个账号的用户。尽管这个用户本身没有做错任何事情，但是他被分配了一个需要为此负责的编号。

别因为这些伪键看上去像是没用的而重新分配它们。

22.3　如何识别反模式

如果你发现团队成员问了如下几个问题，那么他们可能使用了“伪键洁癖”这个反模式。

- “在我回滚了一个插入操作后，要怎么重用那个自动生成的标识？”

  伪键一旦生成后不会回滚。如果非要回滚，RDBMS就必须在一个事务的生命周期内生成一个伪键，而这在多个客户端并发地插入数据时，会导致竞争或者死锁。

- “`bug_id`为4的这条记录怎么了？”

  这是对一个序列的主键中未使用的数字而感到焦虑的表现。

- “我要怎么找到第一个未使用的ID？”

  要这么做的原因几乎肯定就是要重新分配ID了。

- “如果达到了数字标识的最大值怎么办？”

  这通常是重新使用未使用的ID的正当理由。

22.4　合理使用反模式

没有理由改变伪键的值，因为它的值本身并没有什么重要的意义。如果这个主键列有实际的意义，那么这就是一个自然键，而不是伪键。改变自然键的值并不奇怪。

22.5 解决方案：克服心里障碍

主键的值必须是唯一且非空的，因而你才能使用主键来唯一确定一行记录，但这是主键的唯一约束——它们不一定非得是连续值才能用来标记行。

22.5.1 定义行号

大多数伪键返回的数字看起来就像行号一样，因为它们就是依次递增的（每一个新返回的值都比前一个值大1），但这只是由于伪键实现机制所造成的巧合而已。按照这样的方式生成主键值能比较方便地确保唯一性。

别把主键值和行号混为一谈。主键是用来标识表中记录的，而行号是用来标识查询结果集中记录的。查询结果集中的行号和主键没有一丁点关系，尤其是当你使用了JOIN、GROUP BY或者ORDER BY这些操作符的时候。

有很多使用行号的好理由，比如，返回一个查询结果集的子集。通常我们称之为分页，就像在网络搜索时的一页。要选择一个子集，你需要使用到实际的连续增长的行号，但和查询的形式无关。

SQL:2003定义了包括ROW_NUMBER()在内的一些窗口函数，返回一个查询结果集中一段连续的行。通常使用行号的作用是将查询结果限制在一个特定的范围内。

Neat-Freak/soln/row_number.sql

```
SELECT t1.* FROM
  (SELECT a.account_name, b.bug_id, b.summary,
     ROW_NUMBER() OVER (ORDER BY a.account_name, b.date_repor
   FROM Accounts a JOIN Bugs b ON (a.account_id = b.reported_
WHERE t1.rn BETWEEN 51 AND 100;
```

这些函数目前已经被很多主流数据库所支持，包括Oracle、Microsoft SQL Server 2005、IBM DB2、PostgreSQL 8.4和Apache Derby。

MySQL、SQLite、Firebird和Informix还不支持SQL:2003的窗口函数，但它们有一些独有的语法能用来处理本节所描述的问题。MySQL和SQLite提供了一个`LIMIT`子句，Firebird和Informix提供了使用`FIRST`和`SKIP`关键字的查询选项。

22.5.2 使用GUID

当然我们还可以生成随机伪键值，只要你不会重复使用任何数字。有些数据库提供全局唯一标识符（GUID）来达到这个目的。

GUID是一个128位的伪随机数（通常使用32个十六进制字符表示）。由于GUID的定义及其所要实现的目的，它被设计成具有唯一性，因此你可以用其作为伪键。

下面这个例子用的是Microsoft SQL Server 2005的语法：

> Neat-Freak/soln/uniqueidentifier-sql2005.sql

```
CREATE TABLE Bugs (
  bug_id UNIQUEIDENTIFIER DEFAULT NEWID(),
  -- . . .
);

INSERT INTO Bugs (bug_id, summary)
VALUES (DEFAULT, 'crashes when I save');
```

> 整数是不可再生资源吗？
>
> 和伪键洁癖反模式有关的误解是认为定量递增的伪键生成方式最终会用完整数集，未雨绸缪，不能浪费任何一个整数值。
>
> 乍看之下，这么说似乎有点道理。在数学中，整数是一个无限集合，但在数据库中，任何数据类型都只有有限数量的可选值，32位整型最多只能表示232个不同的值。每生成了一个新的主键，离最后一个值就更近一步了。
>
> 我们可以算一下：如果你每天24小时不停地插入新的数据，平均每秒产生1000行记录，用完32位整型所能表示的这些数字总共需要136年。
>
> 如果这还不能满足你的需求，那就使用64位整型。这样，就算你

> 每秒产生一百万条记录，也需要584 542年才能消耗完所有的整型数字。
>
> 所以，要把所有的整型用完几乎是不可能的！

这会生成如下形式的一行记录：

bug_id	summary
0xff19966f868b11d0b42d00c04fc964ff	Crashes when I save

GUID相比传统的伪键生成方法来说，至少有如下两个优势。

- 可以在多个数据库服务器上并发地生成伪键，而不用担心生成同样的值。
- 没有人会再抱怨有断档——他们会忙于抱怨输入32个十六进制字符做主键。

下面这几点是GUID所带来的不便。

- GUID的值太长，不便于输入。
- GUID的值是随机的，因此找不到任何规则或者依靠最大值来判断哪一行是最新插入的。
- GUID的存储需要16字节。这比传统的4字节整型伪键占用更多的空间，并且查询的速度更慢。

22.5.3 最主要的问题

虽然你已经清楚对伪键重新编号和一些相关方案可能会造成的问题，但你还有一个大问题没解决：怎么抵挡一个希望清理数据库中伪键断档的老板的请求？这是一个沟通方面的问题，而不是技术问题。无论如何，你需要让经理理解保护数据库中数据完整性的重要性。

- 向他解释**SQL**的技术。诚实往往是最好的方法。尊重并且理解这个要求之后的原因。比如说，你可以这样告诉经理：

  “断档看上去的确很奇怪，但它们是无害的。在程序运行过程中，有些行被跳过、回滚或者删除都是很正常的。我们每次都

为新插入的行分配一个新的值，而不是写代码来检测哪个老的值能安全使用。这样能使得开发更加简单，执行效率更高，并且能减少错误。”

- 清楚地表明开销。改变主键的值看上去是件小事，但你应该给出关于计算新值的工作量、写代码处理并测试重复值、级联修改整个数据库中的数据、调查对别的系统的影响以及训练用户和管理员使用新的存储过程等事情所需要花费时间的实际估算。

  大多数经理都会根据任务的成本来设定优先级，并且当他们面对真实的开销时，会撤销不合理的请求。

- 使用自然键。如果你的经理或者别的数据库用户坚持认为主键的值是有意义的，那就让主键变得有意义。别使用伪键——使用字符串或者有意义的数字。然后就很容易使用这些自然键所代表的意思来解释产生断档的原因。

  你还可以同时使用伪键和另一个被当成自然标识的属性列。在生成的报告中不显示伪键，从而隐藏会让读者感到不适的断档。

> 将伪键当做行的唯一性标识，但它们不是行号。

第23章　非礼勿视

> 在获得所有证据之前就下结论，是一个重大错误。
>
> 夏洛特·福尔摩斯

“我在你们的产品中找到了另一个Bug。”电话那头说。

20世纪90年代，当我还是一个SQL RDBMS的技术支持时，接到这个电话。我们有一个客户以总是提交一些荒谬的报告而出名。几乎所有他报告的问题最终都证明是他自己犯的小错误，而不是Bug。

“Davis先生，早上好。我们很荣幸能为你解决你所发现的问题。”我回答道：“你能告诉我发生了什么吗？”

“我向数据库发起了一个查询请求，可是什么都没返回。”Davis先生很尖刻地说：“但我确信数据库里有数据，我用一个测试脚本验证过。”

“你的查询有问题吗？”我问道：“API返回任何错误信息了吗？”

Davis回应道：“为什么我要关心API函数的返回值？这个函数就应该执行我的SQL查询。如果它返回错误，就说明你的产品有Bug。如果你的产品没有Bug，就不会返回错误。我的工作不是解决你的Bug。”

我目瞪口呆，但我必须让事实说话才能说服他。“好吧，我们来测试一下。将你的代码中的SQL查询语句复制粘贴到查询工具中，然后执行一下。返回什么？”我等待他回答。

“在`SELECT`附近有语法错误。”他停顿了一下，然后说：“你可以结束这个问题了。”然后他直接挂掉了电话。

Davis先生是一家航空管制公司的开发人员，编写一些软件来记录国际航班的飞行数据。我们每周都要接到几个他的电话。

23.1 目标：写更少的代码

每个人都想写出优雅的代码。也就是说，我们想要用更少的代码来做更酷的事情。同时，我们所做的事情越酷，我们所要写的代码就越少，也越优雅。但如果我们不能让工作变得更酷，至少我们还有足够的理由来改进代码：用更少的代码来提高代码的优雅程度。

这仅仅只是表面上的理由，还有很多别的更靠谱的写出清晰代码的理由。

- 我们可以更快地写完一个程序的代码。
- 我们有更少的代码需要测试、写文档或审查。
- 更少的代码意味着更少的Bug。

因此对于一个程序员来说，删除任何他们能删除的代码是他们的本能，尤其是那些不能让工作看起来更酷的代码。

23.2 反模式：无米之炊

“非礼勿视”对于开发人员来说通常有两种形式：第一，忽略数据库API的返回值；第二，和程序代码混在一起阅读那些分散的SQL语句片段。在这两种情况下，开发人员都会错过那些对他们修复错误非常有帮助的信息。

23.2.1　没有诊断的诊断

See-No-Evil/anti/no-check.php

```
    <?php
①   $pdo = new PDO("mysql:dbname=test;host=db.example.com",
         "dbuser", "dbpassword");
    $sql = "SELECT bug_id, summary, date_reported FROM Bugs
         WHERE assigned_to = ? AND status = ?";
②   $stmt = $dbh->prepare($sql);
③   $stmt->execute(array(1, "OPEN"));
④   $bug = $stmt->fetch();
```

以上的代码非常简洁，但代码中有几处函数返回的状态值可能会引起问题，但如果你忽略了返回值，就永远不会知道怎么回事。

数据库API最有可能返回的错误就是在你建立和数据库之间的链接的时候，比如在代码①处。你可能不小心打错了数据库的名字、服务器的地址、用户名密码，或者数据库服务本身挂了。在实例化一个PDO链接时，会抛出一个异常，可能会直接终止这个范例脚本的执行。

代码②处调用的`prepare()`函数，可能会由于打字错误、不匹配的括号或者拼错了列名导致的语法错误而返回`false`。代码③处，`$stmt`对象的成员函数`execute()`会产生一个致命错误，因为`false`不是一个合法的对象。

```
PHP Fatal error: Call to a member function execute() on a non
```

函数`execute()`也会失败。比如，如果这条查询语句违反了某一个约束，或者超出了访问权限设定，这个函数也会返回`false`。

在代码④处，对函数`fetch()`的调用，在有任何错误产生时，都会返回`false`，比如无法链接到RDBMS。

像Davis先生这样的程序员并不少见。他们可能会觉得对返回值进行

检查或者对异常进行处理对于他们的代码来说没有任何意义，因为他们假设这些可能出错的情况是不可能发生的。同时，这些额外的代码都是重复的，并且让程序看起来很丑陋，而且难以阅读。它们的确一点也不酷。

但用户看不见代码，他们只能看见输出。当一个致命错误没有被处理时，用户就只能看到一个白屏（如图23-1），或者是一个不完整的异常信息。当发生这种情况时，程序代码简短和清晰对用户来说，一点安慰作用都没有。

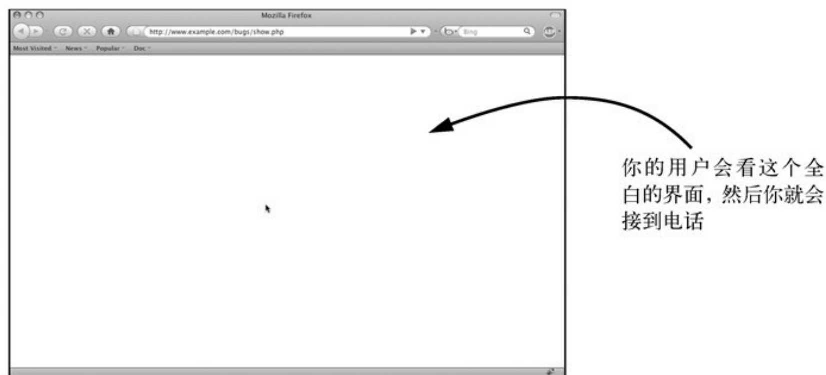


图**23-1**　**PHP**执行中的一个致命错误，导致白屏

23.2.2　字里行间

另一个常见的导致“非礼勿视”这个反模式的坏习惯是，盯着创建SQL查询字符串的代码调试。这么做对于调试来说比较困难，因为用程序逻辑、串连接和应用变量的额外内容建立之后，很难想象出最终拼出来的SQL查询语句到底是什么。这么调试代码就如同不看盒子上的图片直接开始玩拼图一样。

举个简单的例子，让我们看一个我经常听程序员问起的问题。下面的这段代码在断定用户查询一个特定的Bug而不是一个Bug集合后，会将`WHERE`子句连接到这个查询语句的后面，从而为其加上了条件查询的逻辑。

See-No-Evil/anti/white-space.php

```
<?php
```

```
$sql  = "SELECT * FROM Bugs";
if ($bug_id) {
    $sql .= "WHERE bug_id = " . intval($bug_id);
}
$stmt = $pdo->prepare($sql);
```

为什么这个简单的查询会返回错误？如果你看了连接后的$sql变量里的值，真相就会浮出水面：

> See-No-Evil/anti/white-space.sql

```
SELECT * FROM BugsWHERE bug_id = 1234
```

在Bugs和WHERE之间没有空格，造成了查询语句的语法错误。这条语句的意思变成了查询一张叫做BugsWHERE的表，但随后又跟着一串有语法错误的表达式。这段程序代码在连接两个字符串时没有加空格。

开发人员浪费了无数的时间和精力来调试这样简单的问题。通常只要看一眼创建完的SQL语句就能找到问题的症结，但开发人员更喜欢分析创建SQL语句的代码逻辑。

23.3 如何识别反模式

你可能认为，人工定位这些遗漏的错误处理代码很困难，你不见得需要这么做。很多现代的IDE都会在检测到没有处理返回值或者忽略了一个需要检测的异常[1]时，突出显示相应的代码。同时，你还可以根据下面的这些描述，推断出有人使用了“非礼勿视”这个反模式。

> 1 需要检测的异常指的是在函数签名中附带的异常定义，因此使用者可以知道这个函数有可能会返回特定类型的异常。

- “我的程序在我执行查询之后就崩溃了。”

  通常程序崩溃是因为查询失败，然后又试图以一种非法的方式使用返回的结果，诸如调用一个非对象的方法，或者引用一个空指针。

- “你能帮我找一下我的SQL查询中的错误吗？这是我的代码……”

首先，看一下SQL语句，而不是生成它的代码。

- “我不会让错误处理弄乱了我的代码结构的。”

  一些计算机科学家推测在一个稳固的程序中，至少有50%的代码是用来进行错误处理的。这看上去似乎很多，但想想在一个错误处理中所要包含的所有步骤：检查、分类、报告以及补救。对于所有的程序来说，实现错误处理都是很重要的。

23.4 合理使用反模式

当你真的不需要为错误负责的时候，的确可以忽略错误处理。比如，`close()`函数会关闭一个到数据库的链接，然后返回一个状态。但如果你的程序已经运行完成并准备退出了，那所有的链接资源无论如何都会被清空的，也就不需要关心`close()`的返回值了。

在面向对象的语言中，异常允许你跟踪一个错误而不用处理它。任何调用你所提供的接口的代码，才是需要为这个异常负责的代码。因此，你可以直接将一个异常抛出并返回到调用栈的上一层。

23.5 解决方案：优雅地从错误中恢复

所有喜欢跳舞的人都知道，跳错舞步是不可避免的。优雅的秘诀就是弄明白怎么挽回。给自己一个了解错误产生原因的机会，然后就可以快速响应，在任何人注意到你出丑之前，神不知鬼不觉地回到应有的节奏上。

23.5.1 保持节奏

检查数据库API调用的返回状态和异常，是确保你不会跳错舞步的最佳方式。如下的代码对每一个会产生错误的调用都做了检查：

See-No-Evil/soln/check.php

```
    <?php
    try {
        $pdo = new PDO("mysql:dbname=test;host=localhost",
            "dbuser", "dbpassword");
①   } catch (PDOException $e) {
```

```
        report_error($e->getMessage());
        return;
    }

    $sql = "SELECT bug_id, summary, date_reported FROM Bugs
        WHERE assigned_to = ? AND status = ?";

②   if (($stmt = $pdo->prepare($sql)) === false) {
        $error = $pdo->errorInfo();
        report_error($error[2]);
        return;
    }

③   if ($stmt->execute(array(1, "OPEN")) === false) {
        $error = $stmt->errorInfo();
        report_error($error[2]);
        return;
    }

④   if (($bug = $stmt->fetch()) === false) {
        $error = $stmt->errorInfo();
        report_error($error[2]);
        return;
    }
```

代码①处在数据库链接失败时，捕获了其抛出的异常。当有问题的时候，其他的函数就会返回false。

在检查了代码②、③和④之后，你可以从数据库链接对象或者查询语句对象那里获得更多的错误信息。

23.5.2 回溯你的脚步

使用实际的SQL查询语句来进行调试也是很重要的，不能仅仅只是分析产生SQL查询语句的代码。很多简单的错误，诸如拼写错误、引号、括号不全都是非常明显的，即使它们在程序代码中让人费解、困惑。

- 使用一个变量来记录生成的SQL查询语句，而不是在调用API方法，传递参数的时候临时生成。这样做让你有机会在使用生成的SQL语句之前对其进行检查。

- 选择一个不是程序输出的地方输出SQL语句，比如日志文件、

IDE调试控制窗口，或者能显示调试输出的浏览器扩展[2]。

> [2] Firebug（http://getfirebug.com）就是个不错的扩展。

- 别将SQL语句当做HTML注释一起输出在页面中。因为任何人都能看页面源码。阅读这些SQL语句会让黑客了解很多关于数据库架构的信息。

使用一个对象关系映射（ORM）框架来透明地构建和执行SQL查询会使得调试更复杂。如果你不能获得SQL查询的上下文，又怎么观察并且调试问题呢？有些ORM通过将生成的SQL语句写入日志来解决这一矛盾。

最后一点，大多数数据库都提供了它们自己的日志机制，日志是记录在数据库服务器端而非程序客户端的。如果你不能在程序中记录SQL请求，仍可以在数据库服务器端监控这些请求的执行情况。

> 发现并解决代码中的问题已经很困难了，就别再盲目地干了。

第24章　外交豁免权

> 人们讨厌改变。他们总是会说："我们一向如此。"而我尝试改变这一切，这就是为什么我墙上的钟是逆时针的。
>
> 葛丽丝·霍普少将

以前的一份工作曾给我上了重要的一课——关于使用软件工程最佳实践的重要性。那是在一次悲惨的事故之后，我开始接管一个重要的数据库程序。

我受聘于Hewlett-Packard公司，负责开发和维护一个基于UNIX的、使用C和HP ALLBASE/ SQL编写的程序。面试我的经理和员工悲伤地告诉我，他们之前写这个程序的开发人员在一起交通事故中不幸去世了。他们部门中的其他人都不知道如何使用UNIX，也不知道这个程序的任何细节。

我接手之后，发现之前的这个开发人员从来没有写过任何文档或者测试程序，也没有使用任何源代码版本控制程序，甚至连代码注释都没

有。所有的源代码都放在同一个目录下，包括线上运行中系统的代码和开发阶段的代码，以及那些不再使用的代码。

这个项目在技术方面背负重重债务——使用捷径而不是最佳实践的后果。技术债务[1]（technical debt）会不断给项目带来风险和额外的工作，直到你重构、测试并为代码编写文档。

> [1] Ward Cunningham在他关于OOPSLA 1992的报告中创造了这个单词。

我总共花了六个月的时间来重新整理这些代码，并补全相关文档，因为我不得不花很多时间和精力为它的用户和后续开发做技术支持。然而，就程序本身而言，其实真的非常普通。

很显然，我没有办法让我的前任来帮忙加快项目进度。这个项目的经验证明了让技术债务失控所造成的影响有多严重。

24.1　目标：采用最佳实践

专业程序员总是努力地在其项目中使用好的软件工程习惯，比如下面几种。

- 将源代码使用版本控制工具管理起来，比如SVN或Git。
- 为程序编写自动化单元测试脚本或者功能测试脚本。
- 编写文档、规格说明以及代码注释来记录程序的需求和实现机制。

使用最佳实践来开发软件所花费的时间是绝对有益的，因为这么做能减少很多不必要或者重复的工作。大多数资深开发人员都非常清楚地知道，如果为了方便起见而抛开这些实践方法，项目失败就是必然的。

24.2　反模式：将SQL视为二等公民

即使在接受并应用最佳实践的开发人员之中，也有一种思想趋向认为数据库代码是排除在这些实践方法之外的。我将这个反模式命名为“外交豁免权”，因为它假设程序开发的规则并不需要应用到数据库开发中去。

为什么开发人员会做出这样的决定？下面有几个可能的原因。

- 在有些公司里，软件工程师和数据库管理员的角色是分开的，DBA通常同时和好几组程序员一起工作，因此可以说，DBA对于任何一个项目组都不是全职的。DBA被训练成一个参观者，并且不用和软件工程师负同样的责任。

- 使用在关系型数据库中的SQL语言和传统编程语言有很大不同。即使在程序中调用SQL语句的代码，它也看上去和原有代码在风格上格格不入。

- 高级IDE工具在程序编程语言中非常流行，其中编辑、测试和版本控制的功能非常便捷易学。但是数据库开发的工具就不这么高级了，或者使用范围不广。开发人员可以很简单地使用最佳实践来指导编码，但将这些应用到SQL上相比而言就显得很笨拙了。开发人员趋向于，宁可找点别的事情来做，也不会如此尝试。

- 在技术团队，对于数据库的通常认识和做法就是全由一个人来负责——DBA。因为DBA是唯一有权限访问数据库服务器的人，他通常被看做是一个活的资料库和版本控制系统。

数据库是一个程序的基础，和软件质量息息相关。你知道怎么写出高质量的代码，但可能会将程序构建在一个无法解决需求或者没人能看懂的数据库上。这样做的风险就是你在开发一个不得不最终放弃的程序。

24.3　如何识别反模式

你可能认为要证明没做什么很难，但并非每次都是如此。下面就是一些能说明问题的证据。

- “我们正在适应一个新的开发流程：一个轻量级的版本。”

  这里的“轻量级”通常意味着项目组想要跳过一些开发流程中的环节。其中一些可能是合理的，但也可以理解为这是不遵循重要的最佳实践的借口。

- “新的版本控制系统的训练不需要DBA员工的参加，因为他们根本用不到。”

排除一些技术团队的成员参与培训（或者可能都没有访问权限）确保了他们不会使用到这些工具。

- “我要怎么跟踪数据库中的表和列的使用情况？我们不清楚有些条目的作用，还有就是如果它们是孤立的，我们就考虑删除它们了。”

  你没有使用为数据库结构编写的项目文档，或者这份文档过期了，或者无法访问，或者根本就不存在。如果你不知道有些表和列存在的目的，也许它们对别人来说非常重要，你不能随意删除它们。

- “有没有什么工具能比较两个数据库结构，并给出两者的不同之处，然后生成一个将其中一个结构修改成另一个的脚本？”

  如果你没有遵循数据库结构变更部署的流程，它们就可能变得不同步，要将两个数据库的结构改成一致是一项很复杂的工程。

24.4　合理使用反模式

对于要多次使用的代码，我都会编写文档和进行测试，然后将代码置于版本控制之中，同时还有一些别的好习惯来处理这些代码。但我也会写一些即兴的代码，比如对一个API函数写的一次性测试代码以提醒我如何使用这个API，或者回答一些用户提问的代码。

判断代码是否真是临时使用的好方法，就是在你使用完之后立刻删掉它们。如果你不能这么做，那这段代码可能就值得保留下来。同时，这也就意味着这段代码值得存进版本控制系统，并且至少需要写一些简明的注释，说明这段代码的作用以及使用方法。

24.5　解决方案：建立一个质量至上的文化

质量对于大多数程序员来说就是测试，但这仅仅是质量控制——整个流程中的一部分。软件工程的质量保证流程包含如下的三步。

1. 清晰地定义项目需求，并且写成文档。

2. 设计并实现一个解决方案来满足需求。

3. 验证并测试解决方案符合需求。

你需要完成这三步操作才能确保正确的质量保证，虽然在某些软件开发方法中，不需要严格按照上面的这个顺序来执行。

通过遵循编写文档、源码管理和测试的最佳实践，你也可以在数据库开发中保证质量。

24.5.1　陈列A：编写文档

世上没有能够代替文档的代码。尽管一个资深的开发人员能够通过仔细的分析以及凭借自己丰富的经验来解读一段代码，但依旧是非常消耗体力的[2]。同时，代码也不能告诉你还有哪些问题没解决。

> [2] 如果代码是任何人都能读的，那为什么还叫代码呢？

你应该将数据库的需求和实现也写成文档，就像对待程序代码那样。无论你是数据库的原始设计者，还是从别人那里接手的数据库，尽可能地使用下面的清单来检查文档的完整性。

实体关系图。整份数据库文档中最重要的部分，就是一张描述了数据库中所有表及表之间关系的ER图。本书有几章中使用了简单的ER图。更复杂一点的ER图包含了列、主键、索引和其他数据库对象。

不少绘图软件包含了ER图的标记。还有些工具能够通过SQL脚本或者运行中的数据库直接通过反向工程得到ER图。

当数据库过于复杂，表特别多时，要在一张ER图中将所有条目都表示出来会有点不切实际。在这种情况下，你应该将其拆分到多张图中。通常你可以使用“子组”这个符号，这样一来，ER图中即包含了足够的阅读信息，也不至于让人看得晕头转向。

表、列以及视图。你仍需要为数据库编写文档，因为ER图不是描述每张表、列及其他对象的目的和使用方法的正确形式。

对于表来说，需要描述一张表代表了哪种实体类型。比如`Bugs`、`Products`和`Accounts`很容易就能根据表名看出它们的意思，但对于`BugStatus`这种查询表，或者`BugsProducts`这种交叉表，又或者`Comments`这种依赖表来说，很难从表名上看出它们的作用。同时，还需要说明每张表预期存储多少行数据？你希望如何对这张表进行查询？表中有哪些索引？

每个列都有一个列名和对应的数据类型，但这些信息并不能说清楚这个列所代表的含义。比如说，哪些值对于这一列是有意义的（只有很少的情况下，取值范围是对应数据类型的所有可选值的全集）？这个列能否接受NULL，为什么？有没有针对这一列的唯一性约束？如果有，为什么要有？

视图存储了对于一张或多张表的常用查询结果。为什么需要创建这样一个视图？哪个产品或者用户需要使用这个视图？这个视图是不是试图提取表之间的复杂关系？这个视图会不会让没有权限的用户查询一张需要授权的表的部分数据？这个视图是否可以更新？

关系。引用完整性约束表示了表和表之间的依赖关系，但可能并没有完整表达出你所设计的约束模型。比如，Bugs.reported_by是非空的，但Bugs.assigned_to是可空的。这是不是意味着一个Bug可以在指派给任何人之前就被修复？如果不是，一个Bug的指派应该要遵循什么样的规则？

在某些情况下，可能存在一些隐式关系，但没有任何相关的约束。如果没有文档，别人很难知道存在这些关系。

触发器。数据验证、数据转换以及记录数据库变更，都是使用触发器的场景。你在触发器中实现了怎样的业务逻辑？这些都是需要记录在文档中的。

存储过程。要像API那样为存储过程编写文档。这个存储过程要解决什么问题？这个存储过程是否会改变数据？输入输出参数的类型和意义是什么？是否使用这个存储过程来替换一种查询请求，就能够解决某个性能瓶颈？使用这个存储过程是否会让没有权限的用户访问到需要权限的表？

SQL安全。为应用程序指定了哪个数据库用户账号？每个用户都有哪些访问权限？你提供了哪些SQL任务，哪些用户可以使用这些任务？是否有些用户是为了特殊的任务所定义的，比如备份或报表？使用哪种系统安全级别规定，比如客户端必须通过SSL和RDBMS服务器端链接吗？使用何种机制检测和屏蔽非法的认证请求，比如暴力破解密码？有没有针对SQL注入做过一次完整的代码审查？

数据库基础设施：这些信息对运维和DBA的同事来说是最重要的，但开发人员最好也对此有些了解。使用哪个厂商的哪个版本的RDBMS服务？数据库服务器的主机名是什么？是否使用了多台数据库服务器、冗余备份、服务器集群、访问代理等？网络结构是什么样的？数据库

服务器使用的是什么端口？客户端程序连接的时候有什么特殊的选项？数据库的用户密码是什么？数据库的备份方案是什么？

ORM。你的项目可能将一部分应由数据库处理的逻辑写在了程序代码中，作为基于ORM类的层的一部分。哪些业务逻辑是以这样的方式实现的？数据验证、数据转换、日志、缓存还是配置？

开发人员不喜欢维护工程文档。它们既难以编写，也很难保持实时更新，而且当你写了很多却少有人读的时候，会感到很失落。但即使是经验丰富或者极限编程的程序员也知道，他们可以不为程序的其他部分写文档，但一定要编写数据库部分的文档[3]。

> [3] 比如，Jeff Atwood和Joel Spolsky认为，除了数据库部分的文档，其他代码的文档基本没有意义。可以在StackOverflow上看到他们的讨论http://blog.stackoverflow.com/2010/01/podcast-80/。

> 结构演化工具
>
> 版本管理工具管理了代码，但并没有管理数据库。Ruby on Rails提供了一种技术叫做“迁移”，用来将版本控制应用到数据库实例的升级管理上。我们可以简单地来看一个升级的例子。
>
> 可以基于Rails的抽象类来编写一个脚本更新数据库，需写一个能一步完成数据库升级的函数，同时也需要写一个降级函数，能够反转升级函数的操作。
>
> ```
> class AddHoursToBugs < ActiveRecord::Migration
>   def self.up
>     add_column :bugs, :hours, :decimal
>   end
>
>   def self.down
>     remove_column :bugs, :hours
>   end
> end
> ```
>
> Rails“迁移”工具会自动地创建一张表来记录当前数据库实例的多个版本。Rails 2.1引入的修改让该系统更加的灵活，其随后的版

本可能还会改变“迁移”的工作方式。

不断为数据库的每个结构变化创建新的迁移脚本会逐渐积累一系列的脚本，每一个脚本都能对数据库进行一步升级或者降级操作。如果你需要将数据库版本改为5，只需要在运行“迁移”工具的时候指定参数。

```
$ rake db:migrate VERSION=5
```

可以参考“Rails敏捷网站开发，第三版[RTH08]”或者访问 http://guides.rubyonrails.org/migrations.html 来对“迁移”工具进行更深入的学习。

大多数其他的网站开发框架，包括PHP的Doctrine、Python的Django以及微软的ASP.NET，都支持类似于Rails的“迁移”这样的特性，可能集成在框架中，或者以相关项目的形式提供。

“迁移”使很多同步当前数据库实例和源码版本控制服务中指定版本结构的脏活能自动完成。但它们还不是完美的，只能处理一些简单类型的结构变更，而且从根本上说，它们在原有版本控制服务之外又建立了一个版本系统。

24.5.2 寻找证据：源代码版本控制

如果你的数据库服务器彻底挂掉了，要怎么重建一个数据库？跟踪数据库的一个复杂的升级过程的最有效途径是什么？要怎么回滚数据库的变更？

我们知道怎么使用版本控制系统来管理程序代码，解决软件开发中相似的一些问题。一个使用版本控制的项目应该包含在现有程序被损坏时用以重建、部署该项目所需要的一切。版本控制也用来记录历史变更和增量备份，使得你能够回滚任意的修改。

对于数据库的代码，也能使用版本控制，从而获得和程序开发相似的效果。

Diplomatic_immunity/DatabaseTest.php

```
<?php
require_once "PHPUnit/Framework/TestCase.php";
```

```
class DatabaseTest extends PHPUnit_Framework_TestCase
{
    protected $pdo;

    public function setUp()
    {
        $this->pdo = new PDO("mysql:dbname=bugs", "testuser",
    }

    public function testTableFooExists()
    {
        $stmt = $this->pdo->query("SELECT COUNT(*) FROM Bugs"
        $err = $this->pdo->errorInfo();
        $this->assertType("object", $stmt, $err[2]);
        $this->assertEquals("PDOStatement", get_class($stmt))
    }

    public function testTableFooColumnBugIdExists()
    {
        $stmt = $this->pdo->query("SELECT COUNT(bug_id) FROM 
        $err = $this->pdo->errorInfo();
        $this->assertType("object", $stmt, $err[2]);
        $this->assertEquals("PDOStatement", get_class($stmt))
    }

    static public function main()
    {
        $suite = new PHPUnit_Framework_TestSuite(__CLASS__);
        $result = PHPUnit_TextUI_TestRunner::run($suite);
    }

}

DatabaseTest::main();
```

你需要将和数据库开发相关的文件都提交到版本控制服务中去，包括如下的这些文件。

数据定义脚本。所有数据库都提供`CREATE TABLE`或别的定义数据库对象来执行SQL脚本。

触发器和存储过程。很多项目以数据库函数的形式为程序代码提供支持。你的程序可能离开这些函数根本无法工作，因此它们也算做你的

项目代码的一部分。

初始数据。检查表可能包含一些数据，用以在任何用户输入新数据之前初始化数据库。你应该将这些初始数据保存起来，以备需要从项目源码重新构建数据库时使用。初始数据也叫做种子数据。

ER图及文档。这些文件不是代码，但它们和代码、数据库需求、实现机制以及和程序的整合等联系密切。由于项目的升级依赖于数据库和程序的同时升级，你需要将这些文件也保持同步更新。需要确保这些文件切实描述了当前版本的设计。

DBA脚本。大多数项目都有一系列的数据处理任务，这些任务都是在程序之外处理的，包括导入/导出数据、同步数据、生成报表、备份、验证数据、测试等。有些可能是用SQL脚本编写的，而非用一般的编程语言。

确保数据库代码文件是和使用当前数据库的程序代码相关联的。使用版本控制的部分好处就是，当从服务器端签出特定的版本、指定的日期或者不同的里程碑时，所得到的文件应该都能正常工作。你最好将数据库代码和程序代码放在同一个版本库中。

24.5.3　举证：测试

质量保证的最后一步就是质量控制——验证程序做了它应该做的。大多数专业的开发人员都很熟悉编写自动化测试脚本来验证程序代码的行为。测试的一个重要原则就是隔离，即同一时间点只测试系统的一个部分，如果存在缺陷，你可以尽可能缩小出错的范围。

我们在数据库测试中也借鉴这种隔离模块测试的方法，需要独立于程序代码对数据库结构及行为进行验证。

下面的例子展示了使用PHP单元测试框架写的一个单元测试[4]脚本：

> 4 参考 http://www.phpunit.de/。确切地说，测试数据库的功能并不是严格意义上的单元测试，但还是可以使用这个工具来组织和使测试自动化。

你可以在验证数据库的时候参考如下的清单。

表、列和视图。你应该测试一下所希望存在的表、视图是否真的在数据库中存在。每次你使用新的表、视图或者列扩充数据库时，增加一

个确认这些对象存在的新测试任务。你还可以使用负面测试，来验证一张表或者列在当前版本中是否真的删除了。

约束。这是另一个使用负面测试的地方。可以执行INSERT、UPDATE或者DELETE语句来验证约束是否有效，是否正确返回错误。比如，试着违反非空、唯一或者外键约束等。如果所执行的语句没有返回错误，那就意味着对应的约束有问题。你可以通过这样的测试尽早地找到更多的Bug。

触发器。触发器也能进行强制约束。触发器能处理级联效果、转换数据、记录变更等。你应该通过执行一个语句来引发这个触发器测试这些场景，然后再进行一次查询来验证触发器是否按照你所期望的方式执行了。

存储过程。存储过程的测试最接近传统程序代码的单元测试。存储过程有输入参数，如果输入的参数无效则可能会抛出错误。存储过程内部的逻辑可能会有多个分支。存储过程可能会返回单个值，也可能返回一个查询结果集，这取决于输入的参数以及数据库中数据的状态。同时，存储过程也可能会受到数据库正在更新的副作用影响。你可以测试所有这些存储过程的特性。

初始数据。即使理论上完全空白的数据库也会需要一些初始数据，比如检查表中的记录。你可以使用一些查询语句进行查询来验证初始数据是否存在。

查询。程序代码依赖SQL查询。你可以在测试环境中执行一些查询操作来验证语法和结果。确认结果集中包含了所期望的列和数据类型，就像测试表和视图一样。

ORM类。就像触发器一样，ORM类也包含逻辑、验证、转换或者监控。你应该像对待其他的程序代码那样对基于ORM的数据库抽象代码进行测试。确认这些类在输入不同的参数时的行为如期望的那样，并且它们会拒绝接受非法参数。

如果这些测试中的任何一个没有通过，可能是因为程序在使用错误的数据库实例。总是要反复确认你所连接的数据库是否正确——最常见的错误其实就是连错数据库。如果必要，可以修改配置然后重新执行一遍测试。如果你确信链接是正确的，但需要修改数据库，那可以执行一个迁移脚本来同步这个数据库实例到程序所期望的版本。

24.5.4 例证：同时处理多个分支

在开发程序时，你可能需要同时操作多个版本，甚至可能在同一天就操作多个不同的版本。比如，你可能在当前部署的程序分支中修复了一个紧急Bug，然后很快又回到主分支的长期开发中。

但程序所使用的数据库并没有版本控制。要在一眨眼的功夫内重新部署一个数据库是不现实的，即使你所使用的数据库非常敏捷和易于使用。

理想情况下，可以为每个开发、测试、分阶段进行或者部署的程序版本分支创建独立的数据库实例。同时，每个项目组内的开发人员也需要一个独立的数据库实例，从而在不影响整个团队中其他人开发进度的情况下，他们能够完成开发任务。

在程序配置项中增加选择数据库连接的字段，如此一来，无论你在哪个版本下工作，都可以在不修改代码的情况下指定一个数据库链接。

如今，每个RDBMS品牌的商业版和开源版本，都提供开发及测试的免费解决方案。使用诸如VMware Workstation、Xen和VirtualBox之类的平台虚拟化技术，每个程序员都可以以很小的代价运行一个服务器端基础设施的克隆版本，没有理由让程序员在一个和生产环境不一致的环境中进行开发和测试了。

> 数据库和程序都需要采用软件开发的最佳实践，包括文档、测试和版本控制。

第25章　魔豆

> 总会有解释，人类的每一个问题都会有一个众所周知的解决方案——无论是简洁的、可行的，还是错的。
>
> **H. L.** 门肯

“为什么加这么个小功能要花这么长时间？”经理指派你的团队扩展Bug跟踪系统，增加一个展示每个Bug有多少评论数量的功能。你的团队已经为此工作四周了。

在会议室里，你的组员们都羞于回答老板的问题。作为项目经理，你不得不回答：“我们在最开始的时候犯了几个错误，这个需求最初看

起来实现很方便，后来我们意识到在程序中还有其他几个界面也需要展示评论数量。”

“设计这些界面要花四个礼拜？”经理问。

“这倒不是，这些界面只是一些HTML，由于我们用的框架是将代码和展示分离的，这些界面的开发很简单。”你继续说，“但每次我们要往界面上增加一点新的条目，必须复制一份同样的代码到这个界面的后端代码中，用以获取数据。这意味着每个后端的类都需要进行一系列新的测试。”

“我们难道没有使用测试框架？”经理问，“增加一些新的测试用例要花多久？”

“编写测试不像编写代码那么容易，”另一个工程师犹豫地说，“我们还要为编写脚本构建测试数据。接着要在每次测试之前重新将数据载入测试数据库中。我们还要测试前端，每个新增的特性都要针对老功能的每一种组合进行测试。”

经理眼睛瞪得大大的，但你的同事继续说着：“我们现在对于前端已经有600个测试用例了，每一个测试都会创建一个新的浏览器模拟器。时间都花在执行这些测试上了。”他耸耸肩，“我们对此无能为力。”

经理深吸了一口气，说：“好吧……你说的我并不全都理解，我只是想要知道为什么加这么一个简单的功能会变得如此复杂。你们所用的面向对象框架难道不支持快速、简单地添加新功能吗？”

好问题！

25.1　目标：简化MVC的模型

Web程序框架使得往程序中添加新功能变得更简单、快速。在软件项目中，最大的成本就是开发时间。因此，我们所减少的任何一点开发时间，都能使得软件开发的成本降低。

Robert L. Glass认为：“80%的软件工作是智力活动。相当大的比例是创造性的活动。很少是文书性的工作。”[1]

[1] *Facts and Fallacies of Software Engineering*[Gla92]。

有一种方法有助于我们在软件开发过程中思考，那就是采用设计模式的术语和习俗。当我们说单例模式、外观模式或者工厂模式的时候，项目组内的其他开发人员都知道我们在说些什么。这样做节省了很多时间。

在任何项目中，大多数的代码都是重复的——几乎都长得一样。框架通过提供可重用的组件和代码生成工具帮助我们提升编写代码的速度，做到少写代码也能开发出可以工作的软件。

当使用模型—视图—控制器（MVC）架构的时候，我们就是同时在使用设计模式和软件框架。这是一个拆分程序逻辑的技术，就如图25-1所展示的那样。

- 控制器接收用户的输入，定义一些程序需要完成的响应逻辑，再委托合适的模型执行操作，然后将结果返回给视图。
- 模型处理所有其他的事情，它们是程序的核心，包括输入验证、业务逻辑和数据库交互。
- 视图处理在用户界面展示信息。

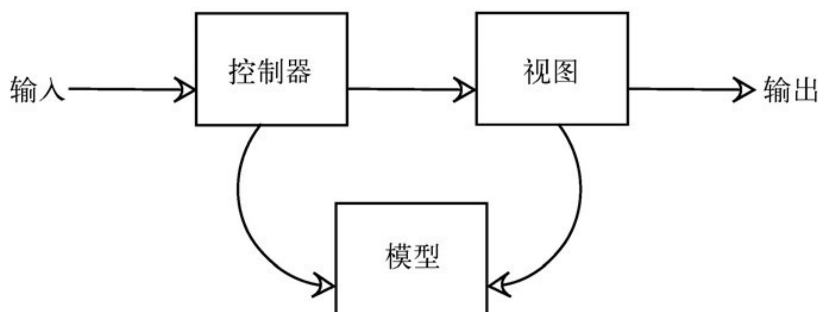


图25-1　模型—视图—控制器（MVC）

理解控制器和视图的行为很容易，但是模型的目的就比较模糊。在软件开发人员的社区中常讨论的问题便是，为了减少软件设计的复杂度，如何简化和抽象模型。但是通常这个目标会导致他们过度简化模型，以至于把它仅仅视作一个数据访问对象。

25.2　反模式：模型仅仅是活动记录

在简单的程序中，你不需要在模型中定义很多逻辑。相对而言，模型

更像是数据库中的某个表的映射对象，也是一种对象—关系映射（ORM）。你需要这个对象做的所有事情是知道如何往表中插入新行、读取一行，以及更新和删除自己——基本的CRUD[2]操作。

> 2 Create, Read, Update Delete，增删改查操作。——译者注

Martin Fowler将这种映射关系概括为一种设计模式，叫做活动记录[3]。活动记录模式是一种数据访问模式。其方法是将一个类与数据库中的一张表或者一个视图相关联。你可以调用这个类的`find()`方法返回一个关联到这张表或者视图的某一行的类对象实例。你还可以使用这个类的构造器来创建一个新行。调用`save()`方法执行插入或者更新现有的行。

> 3 参考*Patterns of Enterprise Application Architecture*中“Active Record”一章。

> Magic-Beans/anti/doctrine.php

```
<?php
$bugsTable = Doctrine_Core::getTable('Bugs');
$bugsTable->find(1234);

$bug = new Bugs();
$bug->summary = "Crashes when I save";
$bug->save();
```

自2004年开始，Ruby on Rails使活动记录模式在Web程序开发框架中流行起来，现在大多数的Web程序框架使用这种模式作为数据访问对象（DAO）。使用活动记录模式并没有什么错，这是一个很好的模式，提供了简单访问表中特定行的接口。我们所要说明的反模式是在MVC程序中，所有模型类都继承自同一个活动记录基类这一习惯做法。这是一个“金锤子”反模式的例子：如果你的唯一工具是一把锤子，那么就会将每个东西都看做钉子。

所有能简化软件设计的做法都很吸引人。如果我们愿意牺牲一些灵活性，就能让工作更简单，而且，如果灵活性从一开始就不重要，那就更好了。

但这是个童话故事，就像《杰克与魔豆》一样。杰克相信当他睡着的时候，他的魔豆会长成一个巨大的豆茎。在杰克的故事中的确就是这么发生的，但我们不可能都这么幸运。让我们来看看使用魔豆反模式的后果。

> 抽象泄露
>
> Joel Spolsky在2002年提出了“抽象泄露”这个词。*抽象简化了一些技术的内部工作原理并且让其更加方便调用。但当由于需要更高效的生产效率而不得不了解内部细节的时候，就称之为抽象泄露。
>
> 在MVC架构中，把活动记录模式作为模型层来使用就是抽象泄露的例子。在非常简单的项目中，活动记录模式就像魔法一样神奇。但如果你想要在所有的数据库访问上都使用这个模式，你就会发现很多诸如`JOIN`或者`GROUP BY`的这些在SQL中很简单的操作，在活动记录模式中变得很可怕。
>
> 有些框架尝试扩展活动记录模式来支持更大规模的SQL语句。但是框架暴露的使用SQL内部接口越多，你就会越觉得不如直接使用SQL来得方便。
>
> 抽象模式因此不再能隐藏它的秘密，就像绿野仙踪里的托托发现精灵不过是藏在窗帘后的普通人一样。
>
> ---
>
> * 参考《抽象泄露法则》[Spo02]

25.2.1　活动记录模式连接程序模型和数据库结构

活动记录模式是一种简单的模式，因为一个普通的活动记录类只能表示数据库中一张表或者一个视图。活动记录对象中的每一个字段对应于相关表中的一列。如果你有16张表，就要定义16个模型子类。

这意味着，如果需要重构数据库来表示一个新数据结构，你的模型类就需要跟着改变，同时，任何使用这些模型类的代码也都需要改变。还有，如果增加了一个控制器来处理新的程序界面，你也需要重复这些查询模型的代码。

25.2.2　活动记录模式暴露了CRUD系列函数

接下来你需要面临的问题就是，其他使用模型类的程序员会无视你设定的使用方法，他们会直接使用CRUD函数来更新数据。

比如，你可能在Bug模型中有一个叫做`assignUser()`的方法解决每次更新Bug之后发送一封邮件给相关工程师的需求。

Magic-Beans/anti/crud.php

```
<?php
class CustomBugs extends BaseBugs
{
  public function assignUser(Accounts $a)
  {
    $this->assigned_to = $a->account_id;
    $this->save();
    mail($a->email, "Assigned bug",
      "You are now responsible for bug #{$this->bug_id}.");
  }
}
```

然而，另一个编程人员忽略了你的方法，他手动地分配了这个Bug却没有发送邮件。

Magic-Beans/anti/crud.php

```
$bugsTable = Doctrine_Core::getTable('Bugs');
$bugsTable->find(1234);
$bug->assigned_to = $user->account_id;
$bug->save();
```

你的需求是无论何时对一个任务进行变更操作，都要有一封邮件提醒。而这种模型设计允许略过这一步。将活动记录类的CRUD方法暴露给驱动模型类是否真的有意义呢？要如何阻止那些使用这些方法的程序员的不合理调用？如何在编写项目文档和具体代码实现的时候，将这些活动记录的接口从你的模型类中排除呢？

25.2.3　活动记录模式支持弱域模型

另一个非常值得关注的问题，就是一个活动记录模型通常除了CRUD方法之外，没有别的行为。很多程序员扩展了这个基本的活动记录

类，却不为这个模型所需处理的业务逻辑添加新的方法。

将模型简单地当成数据访问对象，鼓励开发人员将业务逻辑放在模型类之外去实现，通常这些逻辑就会被拆分到多个控制类中，从而减少了模型本身的内聚行为。Martin Fowler在他的博客里称这种反模式为弱域模型4。比如说，你可能有一系列独立的活动记录类，它们分别关联到Bugs、Accounts和Products表，但在很多地方你都是同时需要这三张表中的数据的。

> 4 http://www.martinfowler.com/bliki/AnemicDomainModel.html。

让我们看一段Bug跟踪程序中的代码，其简单地实现了Bug指派、数据输入、Bug显示和搜索的工作。它使用的PHP框架叫做Doctrine，这个框架提供简单的活动记录接口，并且使用Zend框架来实现MVC架构。

> Magic-Beans/anti/anemic.php

```
<?php
class AdminController extends Zend_Controller_Action
{
  public function assignAction()
{
    $bugsTable = Doctrine_Core::getTable("Bugs");
    $bug = $bugsTable->find($_POST["bug_id"]);
    $bug->Products[] = $_POST["product_id"];
    $bug->assigned_to = $_POST["user_assigned_to"];
    $bug->save();
  }
}

class BugController extends Zend_Controller_Action
{
  public function enterAction()
  {
    $bug = new Bugs();
    $bug->summary = $_POST["summary"];
    $bug->description = $_POST["summary"];
    $bug->status = "NEW";

    $accountsTable = Doctrine_Core::getTable("Accounts");
```

```
    $auth = Zend_Auth::getInstance();
    if ($auth && $auth->hasIdentity()) {
      $bug->reported_by = $auth->getIdentity();
    }
    $bug->save();
  }

  public function displayAction()
  {
    $bugsTable = Doctrine_Core::getTable("Bugs");
    $this->view->bug = $bugsTable->find($_GET["bug_id"]);

    $accountsTable = Doctrine_Core::getTable("Accounts");
    $this->view->reportedBy = $accountsTable->find($bug->repo
    $this->view->assignedTo = $accountsTable->find($bug->assi
    $this->view->verifiedBy = $accountsTable->find($bug->veri

    $productsTable = Doctrine_Core::getTable("Products");
    $this->view->products = $bug->Products;
  }
}

class SearchController extends Zend_Controller_Action
{
  public function bugsAction()
  {
    $q = Doctrine_Query::create()
      ->from("Bugs b")
      ->join("b.Products p")
      ->where("b.status = ?", $_GET["status"])
      ->andWhere("MATCH(b.summary, b.description) AGAINST (?)
    $this->view->searchResults = $q->fetchArray();
  }
}
```

使用活动记录的控制器代码逐渐变成组织程序逻辑的技术手段。如果数据库的结构或者程序的期望行为改变了，你就需要更新代码中的很多地方。同时，如果增加了一个控制器，即使这个控制器对模型对象的查询逻辑和其他的控制器中的实现很类似，你也要编写新的代码来处理。

类交互图（图25-2）很混乱而且难以阅读，当增加了新的控制器和DAO类时，它只会变得更糟糕。这是一个很强烈的信号——这些同时

使用不同模型的代码在多个控制器复制来复制去，你需要使用另一种方法来简化和压缩程序。

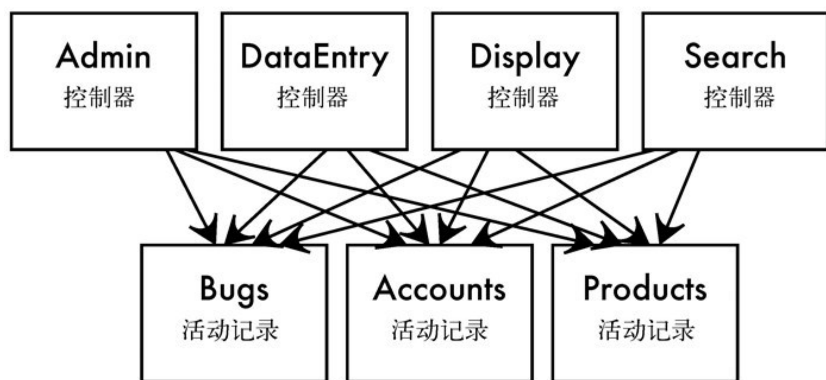


图25-2　使用魔豆造成混乱

25.2.4　魔豆难以进行单元测试

使用了魔豆反模式，你会发现测试MVC的每一层都变得更加困难。

- 测试模型：由于将模型类等同于活动记录类，你无法将数据访问与模型行为分开测试。要测试这些模型，你就必须连上真实的数据库执行查询。

  很多人使用数据库固定工具。数据库固定工具将数据载入到测试数据库中，来确保每个测试使用的是同样的标准数据。这样复杂的步骤使得测试模型的过程变得缓慢且容易出错，就和请求真实数据库进行测试一样麻烦。

- 测试视图：测试视图包括将视图呈现成HTML内容并解析其结果，来验证由模型提供的动态HTML条目确实出现在了输出中。即使你所使用的框架简化了测试脚本中的判定过程，框架还是要执行复杂且耗时的代码来呈现及解析HTML中指定的条目。

- 测试控制器：你将发现测试控制器也变得很复杂，因为模型就是导致同样的代码在多个控制器中重复出现的数据访问对象，所有的代码都需要测试。

  要测试控制器，你需要模拟一个HTTP请求。Web程序的输出是

一个HTTP响应包的包头和包体。要验证这个测试的结果，就不得不拆分由控制器返回的HTTP响应的内容。这需要很多的代码来测试业务逻辑，导致测试进行得很慢。

如果你能够将业务逻辑和数据库访问以及展示层拆分开，将有助于达成MVC的目标，同时也会让测试变得更简单。

25.3　如何识别反模式

下面的线索可能意味着你用了魔豆反模式。

- “我要怎么传递一个自定义的SQL查询给模型？”

  这个问题表示了你正在使用一个数据库访问类做为模型类。你不应该将SQL查询语句传递给模型对象——模型类应该囊括了所有它需要的查询。

- “我应该复制复杂的模型查询到我所有的控制器内，还是在一个抽象控制器内实现这些代码？”

  这两种方案都无法给你带来简单性或者稳定性。你应该将复杂的查询代码写在模型类里面，作为模型类的接口暴露出来。这样，就遵循了“不要重复自己（DRY[5]）”的原则，并且模型使用起来更简单。

  > [5] DRY出现在*The Pragmatic Programmer*[HT00]，由Andy Hunt和Dave Thomas所著。

- “我不得不写更多的数据库固定工具来对模型进行单元测试。”

  如果你正在使用数据库固定工具，就是说正在测试数据库访问而不是业务逻辑。你应该将对模型的单元测试和数据库分离开。

25.4　合理使用反模式

本质上说，活动记录模式并没有什么错，对于简单的CRUD操作来说，这是一个很方便的模式。在大多数程序中，总有一些情况只需要简单的数据访问对象，来提供一些简单的对于表的行操作。此时，你可将模型和DAO视作同一个东西。

另一个使用活动记录的好地方是原型开发。当快速编码比写代码的可测试性和可维护性还重要的时候，捷径是关键。在早期的工作中展示一个原型来获得积极的反馈，通常是提炼项目的一个好方法。任何你能加速原型开发的方法在这些情况下都是很有帮助的，使用一个简单的程序框架也很有用。

此外，你必须要确认你留有一定的时间来重构代码，从而偿还在原型开发阶段的技术债务。

25.5 解决方案：模型包含活动记录

控制器处理程序输入，视图处理程序输出，两者的任务都相对简单且定义清楚。框架是帮助你将这些东西快速融合在一起的最好方法。但是对于框架来说，很难给模型提供一个“通吃”的解决方案，因为模型包含了面向对象设计中的其余部分。

这是你确实需要仔细考虑的部分，关于你的程序中的对象是什么，以及这些对象包含什么数据和行为。还记得Robert L. Glass评估软件开发中的主要工作是智力活动和创造力吗？

25.5.1 领会模型的意义

所幸的是，在面向对象设计领域，有很多格言警句能指导你。比如，Graig Larman的《UML和模式应用》（*Applying UML and Patterns*[Lar04]）一书，描述了很多指导方针，被称为通用职责软件分配模式（GRASP）。其中的一些指导原则和分离模型和数据访问对象尤其相关。

信息专家

一个对象应该有所有需要的数据来满足它所负责的操作。由于程序中的一些操作可能会包含多个表（或者没有表），并且活动记录只适用于一次操作一张表，我们需要另一个类来聚合多个数据库访问对象，并利用这些对象进行组合操作。

模型和活动记录之类的DAO之间的关系应该是HAS-A（聚合）而不是IS-A（继承）。大多数依赖于活动记录模式的框架都使用IS-A的解决方案。如果你的模型使用DAO而不是直接从DAO类继承，那么你就能将这个模型设计成包含所有的数据和代码来表达它代表的领域的形式——即使需要用多张表来表示。

创造者

一个模型如何维护数据库应该是它的内部实现细节，一个聚集了一系列DAO的领域模型应该负责创建这些对象。

程序的控制器和视图应该使用领域模型的接口，而不用关心这个模型使用哪种数据库交互方式对数据进行存取。这使得日后修改数据库查询变得更加容易，只需要修改一个地方。

低耦合

解耦程序中的逻辑区块是非常重要的，这么做能够让你更加灵活地改变某一个类的实现，同时又不影响这个类的调用方式。程序的需求是无法简化的，一些复杂的逻辑无法在程序中舍弃，但你可以对在何处实现这些复杂的逻辑做出正确的选择。

高内聚

领域模型的接口应该反映出它所期望的调用方式，而不是数据库物理结构或者CRUD操作。通常的活动记录模式的接口，如`find()`、`first()`、`insert()`或者`save()`，并没有给出多少对软件需求实现方面的信息。而类似于`assignUser()`的方法则更加具有说明性，并且控制器代码也能变得更容易理解。

将模型类和它所使用的DAO解耦后，你甚至可以为同一个DAO设计多个模型类。这比将所有关于给定表的相关工作都合并到单一展开的活动记录类中要好很多。

25.5.2　将领域模型应用到实际工作中

在《领域驱动设计：软件核心复杂性应对之道》（*Domain-Driven Design：Tackling Complexity in the Heart of Software*[Eva 03]）一书中，Eric Evans介绍了一个更好地解决方案：领域模型。

在最初的MVC概念中（而不是武断的软件设计），一个模型所表示的是程序中某一领域的面向对象的表现手段，也就是说，程序中的业务逻辑和所需要的数据。模型是实现程序业务逻辑的地方，将数据存储在数据库中是模型的内部实现细节。

既然我们让模型设计围绕着程序的逻辑，而不是数据库层面，就可以将数据库操作完全隐藏在模型类里面。看一下可以重构我们早先的例

子的几处地方：

Magic-Beans/soln/domainmodel.php

```
<?php

class BugReport
{
  protected $bugsTable;
  protected $accountsTable;
  protected $productsTable;

  public function __construct()
  {
    $this->bugsTable = Doctrine_Core::getTable("Bugs");
    $this->accountsTable = Doctrine_Core::getTable("Accounts"
    $this->productsTable = Doctrine_Core::getTable("Products"
  }

  public function create($summary, $description, $reportedBy)
  {
    $bug = new Bugs();
    $bug->summary = $summary
    $bug->description = $description
    $bug->status = "NEW";
    $bug->reported_by = $reportedBy;
    $bug->save();
  }

  public function assignUser($bugId, $assignedTo)
  {
    $bug = $bugsTable->find($bugId);
    $bug->assigned_to = $assignedTo"];
    $bug->save();
  }
  public function get($bugId)
  {
    return $bugsTable->find($bugId);
  }

  public function search($status, $searchString)
  {
    $q = Doctrine_Query::create()
```

```
      ->from("Bugs b")
      ->join("b.Products p")
      ->where("b.status = ?", $status)
      ->andWhere("MATCH(b.summary, b.description) AGAINST (?)
    return $q->fetchArray();
  }
}

class AdminController extends Zend_Controller_Action
{
  public function assignAction()
  {
    $this->bugReport->assignUser(
      $this->_getParam("bug"),
      $this->_getParam("user"));
  }
}

class BugController extends Zend_Controller_Action
{
  public function enterAction()
  {
    $auth = Zend_Auth::getInstance();
    if ($auth && $auth->hasIdentity()) {
      $identity = $auth->getIdentity();
    }
    $this->bugReport->create(
      $this->_getParam("summary"),
      $this->_getParam("description"),
      $identity);
  }

  public function displayAction()
  {
    $this->view->bug = $this->bugReport->get(
      $this->_getParam("bug"));
  }
}
class SearchController extends Zend_Controller_Action
{
  public function bugsAction()
  {
    $this->view->searchResults = $this->bugReport->search(
      $this->_getParam("status", "OPEN"),
```

```
        $this->_getParam("search"));
    }
}
```

你可能注意到了几处改进。

- 类交互图（图25-3）变得更加简单且易读了，象征着各个类之间的解耦。
- 通过解耦模型接口和底层的数据库结构，我们减少且简化了控制器的代码。
- 模型类创建对象和一个或者多个表进行交互，控制器不需要知道哪张表被引用了。
- 模型类封装、隐藏了数据库查询，控制器只关心用户的输入，然后通过调用模型的API来执行更高层次的任务。
- 某些情况下，一个查询太复杂而无法简单地使用一个DAO，编写自定义的SQL就变得很必要。SQL安全地包含在模型类里时，直白地使用它看上去不那么可怕。

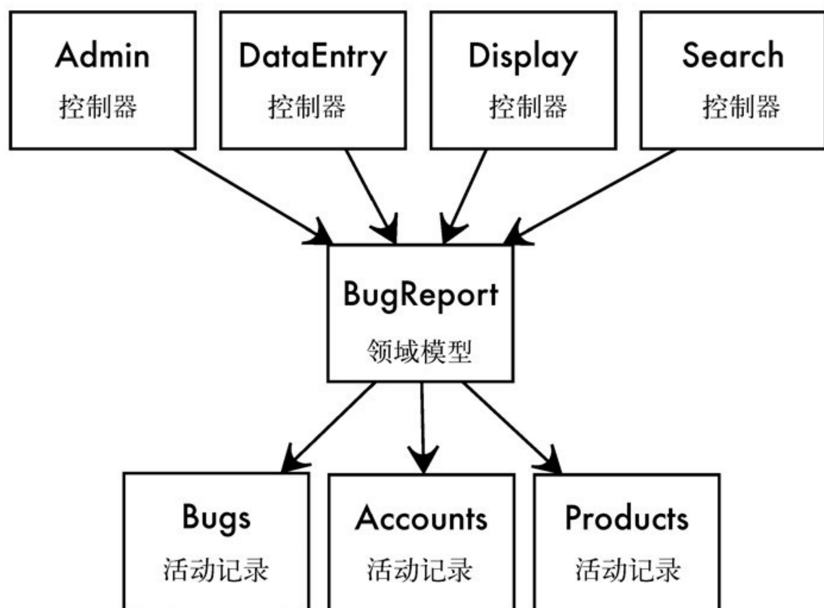


图25-3 通过解耦减少混乱

25.5.3 测试简单对象

理想情况下，你可以不用连接到真实的数据库来测试模型。如果将模型和DAO解耦，那么可以模拟DAO来帮助对模型进行单元测试。

同样，你可以像其他面向对象的测试那样测试领域模型的接口：调用对象的方法，然后验证这个方法的返回值。这比模拟HTTP请求来填充一个控制器，并且解析HTTP的响应要快得多，容易得多。

你还是需要模拟HTTP请求来测试控制器，但由于控制器的代码变得更简单，所以不需要测试很多逻辑分支。

如果将模型和控制器、数据组件分离，就可以对所有这些类都进行更简单的、更独立的单元测试。这能帮助你更容易地定位缺陷。这可就是单元测试的目的所在啊！

25.5.4 回到地球

你可以在任何软件开发框架中更有效地使用数据访问对象，即使这个框架鼓励使用魔豆反模式。然而，那些不知道如何使用面向对象设计原则的开发人员，注定会写出意大利面条式的代码。

本章所提及和描述的领域模型的基本概念，能帮助你选择最好的设计来支持测试和代码维护，最终会达到高效开发基于数据库的程序的目的。

> 将模型和表解耦。

第五部分　附录

本部分内容

- 附录A　规范化规则
- 附录B　参考书目

附录A　规范化规则

> 年轻人，在数学里，你们并不理解事情，你们只是去习惯。
>
> 约翰·冯·诺伊曼

设计关系型数据库的过程既不靠臆断，也不神秘。你可以使用定义好的一系列规则来设计数据存储策略，从而避免冗余并防止应用程序出错，就像本书之前提到的poka-yoke方法一样。你可能听说过类似的一些概念，比如“防御设计”或者“尽早暴露错误”等。

规范化规则并不复杂，但很隐晦。开发人员通常会误解它们的原理，可能是因为他们将规则想得过于复杂了。

另一种可能性就是开发人员本身抗拒遵循规则。规则是那些具有创新精神的程序员所鄙视的东西，也是自由的对立面。

软件开发人员需要不断在简单和灵活之间进行取舍。你可以做很多重新发明轮子的工作和开发自定义的数据管理软件，或者通过遵循相关的设计，利用已有的知识和技术。

本书中我根据那些反模式的优点（或者缺点）来描述它们，来避免过于学术或者理论性。但在这个附录中，我们将看到理论也可以变得很实用。

A.1　关系是什么

“关系”这个术语并不是指表和表之间的关系。它指表自身，或者更进

一步，是指表中列之间的关系。某种程度上来说，它也同时表示这两种关系。

数学家将关系定义为，两个不同数据域上的值的集合通过一定的条件得到一个所有可能组合的子集。

比如，一个包含所有棒球队名字的集合，和一个包含所有城市的集合。将每个城市和球队的组合都列出来，这个列表可以很长很长。但我们只关注这个列表的一个子集：球队和其所属城市的组合。有效的组合包括Chicago/White Sox、Chicago/Cubs或者Boston/Red Sox，但没有Miami/Red Sox。

“关系”这个词有两种用法：作为一种规则（“城市指某一支队伍所属的城市”），或者作为过滤子集的规则。在SQL中，我们可以将这个子集存储在一张有两列的表中，每一行对应一个组合。

当然，关系并不局限于只有两列，可以将任意数量的数据域，每个占一列，合并在一起组成一个关系。同时，数据域可以是32位整型数的集合，也可以是固定长度字符串的集合。

在对表进行规范化处理之前，需要确认它们的关系是合适的，它们必须满足一些条件。

A.1.1　行之间没有上下顺序

在SQL中，查询返回的结果的顺序是不定的，除非使用`ORDER BY`指定排序规则。当然，除了顺序，结果集中的数据总是一致的。

A.1.2　列之间没有左右顺序

无论是让Steven测试Open RoundFile这个项目的第1234号Bug，还是我们需要知道Open RoundFile的第1234号Bug是否能让Steven来验证，最终得到的查询结果应该是一样的。

这和第19章的反模式相关，那时我们使用列的顺序而不是列名。

A.1.3　重复行是不允许的

对于一个事实，进行更多的证明也不会让它变得更正确。给定一个棒球队的队名，数据就能指出它所在的城市是哪个，我们可以说城市依赖于队名。

要阻止重复记录，我们必须能区别两行数据，并且能定位指定的行。在SQL中，我们对列或者列的集合使用主键约束来确保这一点，而不管是否需要保证记录唯一性。

在其他非主键列里可能还是存在重复——波士顿有两支球队——但整体来看每一行是不重复的，因为这两支队伍的队名不同。

A.1.4　每一列只有一种类型，每一行只有一个值

关系头定义了列的名字和数据类型。每一行数据拥有的列和头中的定义必须匹配，且每一列的意义在所有行中必须一致。

第6章的反模式有两处违反这个规则的地方。首先，EAV的表让每个实例的模型都可以自定义属性集，因此，实体的结构和任何头定义都不一样。

其次，EAV的`attr_value`列包含了所有实体的属性，包括Bug的报告日期、Bug的状态、Bug被指派给哪个账户等。1234这个值可能对于两个不同的属性来说都是合法的，但又完全具有不同的含义。

第7章中的反模式也破坏了这条规则，由于1234这个值可能会引用任意多个父表中的主键，因此无法断定不同两行中的1234是否具有相同的含义。

A.1.5　行没有隐藏组件

列中存储的是数据的值，不包含物理存储标示，诸如行ID或者对象ID。在第22章中，我们知道主键是唯一的，但并不是实际的行号。

有些数据库绕开了这条规则，提供了访问数据库内部存储细节的扩展SQL语法（比如，Oracle的`ROWNUM`伪列，或者PostgreSQL中的`OID`列）。然而，这些数据并不属于关系结构。

A.2　规范化的神话

很难再找出一个像规范化这种即使有精确定义依旧被广泛误解的主题。实际生活中，你肯定遇到过相信下面这些错误理念的开发人员。

- “规范化让数据库更慢。不规范让数据库更快。”

错！的确，应用了规范化之后，查询时可能需要使用JOIN从多张分开的表中获取数据，而不规范的数据能够避免这些JOIN。

比如，在第2章中使用逗号分割的列表来获取Bug相关的产品。但如果我们同时还需要根据指定的产品获取所有相关的Bug呢？非规范化通常能简化或者提升某种类型的查询，但应用在别的查询类型上时开销就很大。

非规范化也有合理使用的场景，但是在决定使用它时，应该先将数据库设计成标准的形式。第13章中的MENTOR规则也使用了非规范化。请记住，如果是为了性能而做的修改，则必须在修改前后都进行性能测量。

- “规范化也就是说将数据移到子表中，然后使用伪键引用它们。”

  错！你可以为了方便、性能或者存储效益等所有合理的理由而使用伪键，但别认为这和规范化有什么关系。

- “规范化就是将属性尽可能地拆分开，就像EAV设计那样。”

  错！通常程序员会误解“规范化”这个词，认为它把数据弄得更不可读或者不便于查询。而事实上，真正的“规范化”正好与之相反。

- “没人需要超过第三范式的规范化标准。其他的范式都太晦涩难懂，并且你永远不会用到它们的。”

  错！调查表明，超过20%的商业数据库的设计满足第一到第三范式，但违背第四范式。虽然这个量看上去比较小，但并不能说它就可以忽略：如果20%的程序中存在潜在的Bug会导致数据丢失，你会不想解决它吗？

A.3　什么是规范化

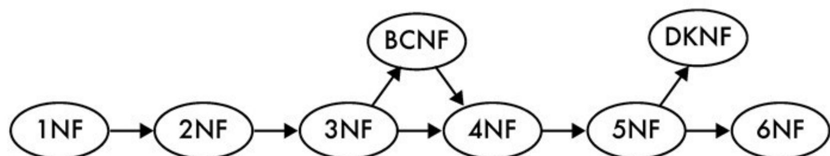
下面这些是规范化的目标：

- 以一种我们能够理解的方式表达这个世界中的事物；
- 减少数据的冗余存储，防止异常或者不一致的数据；

- 支持完整性约束。

请注意，提高数据的性能并不在此列表中。规范化有助于我们正确地存储数据，避免陷入麻烦。当数据库是非规范化的时候，几乎不可避免地会变成一个烂摊子，我们需要编写更多的代码来清理不一致的或者重复的数据，最终我们会因为错误数据而不得不延期项目以及投入更多的成本。如果将所有这些场景考虑进去，就更容易看出规范化所带来的效率提升。

当一张表满足规范化的规则时，我们便称这张表符合范式。有五种传统的范式，描述了依次递进的规范化等级。每种范式消除了一种特定类型的冗余或者异常情况。通常来说，如果一张表的设计满足某一层级的范式，那这张表一定满足前面所有层级的范式。研究人员还定义了这五种传统范式之外的另外三种范式。范式的渐进级别如图A-1所示。



图**A-1**　范式渐进级别

A.3.1　第一范式

第一范式的最根本要求是，该表必须是一个关系。如果它不符合A.1节中所描述的关系准则，那么这张表就不符合第一范式和后续的范式。

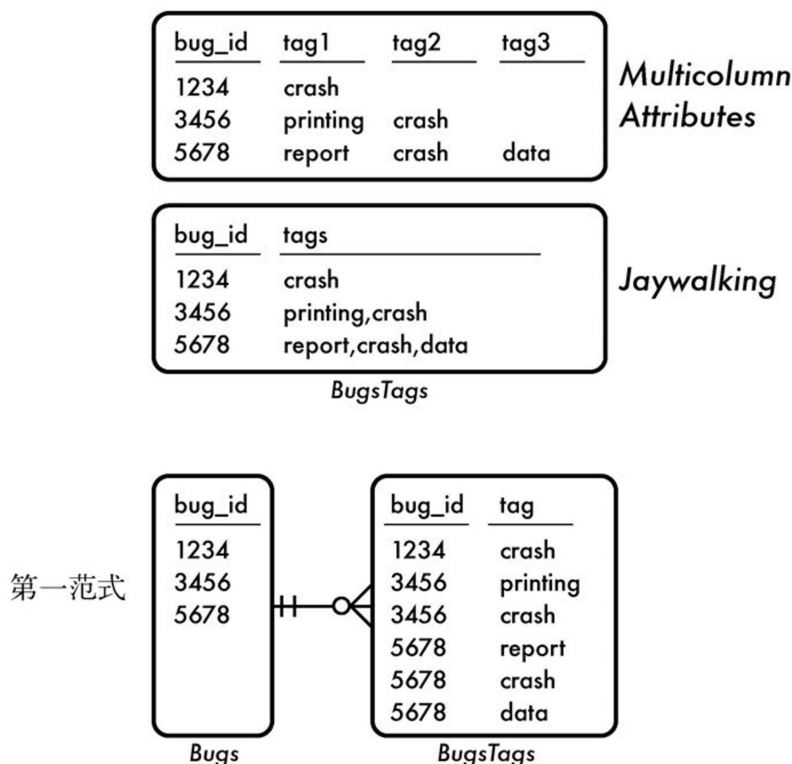
接下来的要求是这张表必须没有重复组合。请记住，关系中的每一行，都是从多个集合的每一个集合中选一个值形成的一种组合。重复的组合说明这一行可能有多个来自于同一个集合的值。

我们曾经见过两个创建了重复组合的反模式。

- 第8章在多个列中出现了来自同一个域的值。
- 第2章在同一列中有多个值。

从图A-2中可以看到这两个反模式中的重复组合。满足第一范式的更合理的设计应该是创建一个单独的表，tag占用单独的一列，并且通

过每行存储一个标签来支持多个标签的存储。



图A-2 重复组合与第一范式

A.3.2 第二范式

除了复合主键之外，第二范式和第一范式是一样的。在之前的标签例子中，我们保持跟踪哪些用户给Bug打上了特定标签，我们还要关注是谁第一个创造了某一个标签。

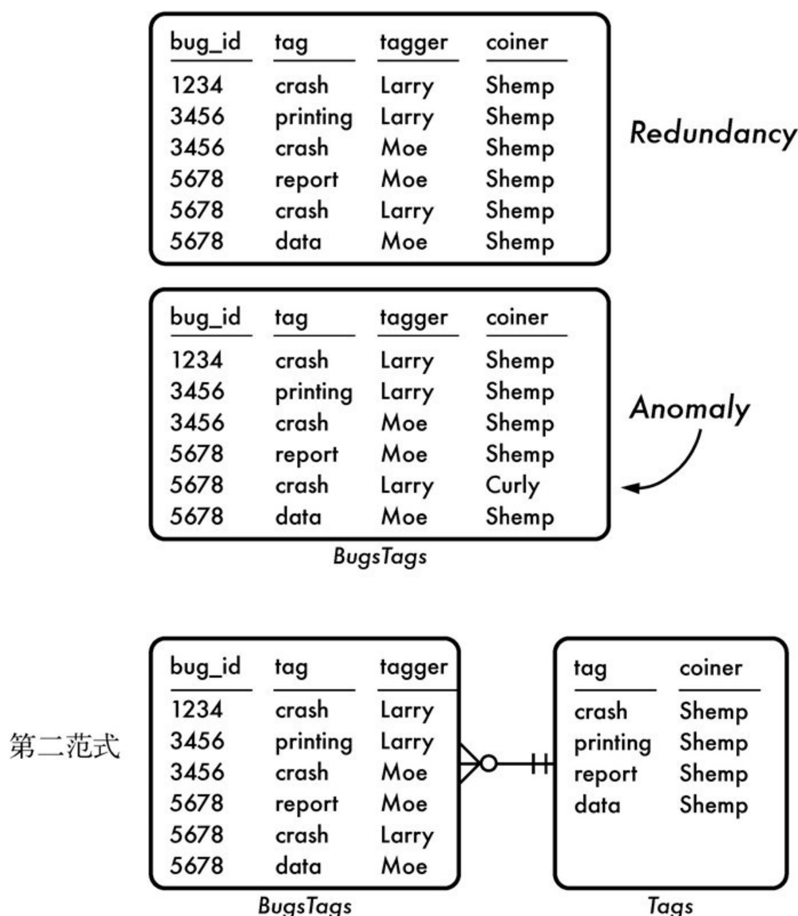
Normalization/2NF-anti.sql

```
CREATE TABLE BugsTags (
  bug_id  BIGINT NOT NULL,
  tag     VARCHAR(20) NOT NULL,
  tagger  BIGINT NOT NULL,
  coiner  BIGINT NOT NULL,
```

```
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (tagger) REFERENCES Accounts(account_id),
  FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);
```

在图A-3中，可以看到创造者标识的存储是冗余的。[1]这意味着修改了某一个tag（如crash）的创造者的标识，如果没有同步所有相同标签的行，则会导致数据异常。

> [1] 此图使用用户名字而非ID号来标识用户。



图**A-3**　冗余与第二范式

为了满足第二范式，我们应该对于每个tag只存储一次它的创造者。也就是说，我们不得不额外定义一张表Tags，以tag作为主键，这样每一个tag就只有一行了。接着，我们就可以将tag的创造者从BugsTags表里移到这张表中，从而防止了异常发生。

Normalization/2NF-normal.sql

```
CREATE TABLE Tags (
  tag      VARCHAR(20) PRIMARY KEY,
  coiner   BIGINT NOT NULL,
  FOREIGN KEY (coiner) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsTags (
  bug_id   BIGINT NOT NULL,
  tag      VARCHAR(20) NOT NULL,
  tagger   BIGINT NOT NULL,
  PRIMARY KEY (bug_id, tag),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (tag) REFERENCES Tags(tag),
  FOREIGN KEY (tagger) REFERENCES Accounts(account_id)
);
```

A.3.3　第三范式

在Bugs这张表中，可能需要记录处理Bug的工程师的E-mail。

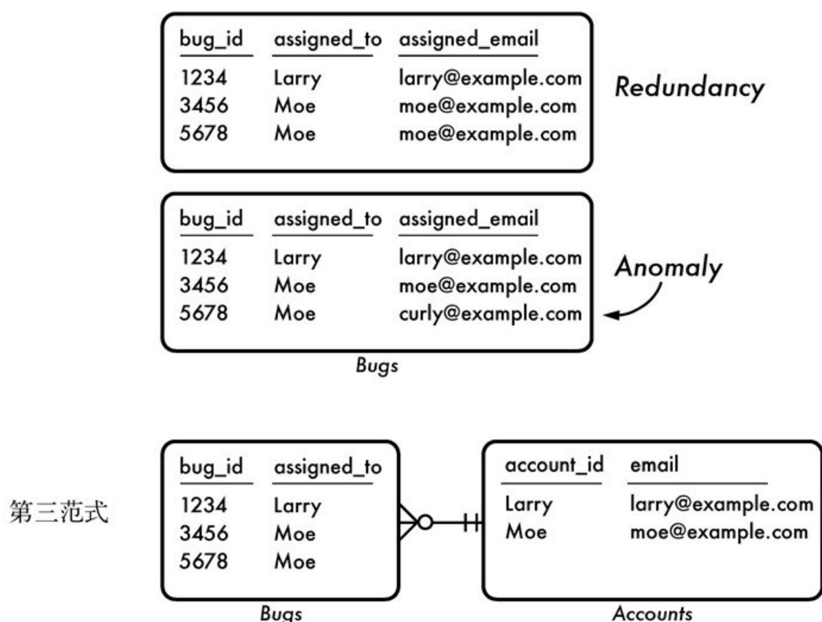
Normalization/3NF-anti.sql

```
CREATE TABLE Bugs (
  bug_id SERIAL PRIMARY KEY
  -- . . .
  assigned_to BIGINT,
  assigned_email VARCHAR(100),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);
```

然而，E-mail是这个被分配任务的工程师账号的一个属性，并不是Bug的属性。以这种形式存储E-mail就是冗余的，并且我们需要面对之前不满足第二范式时产生的那种风险。

第二范式例子中的那个犯规的列至少还部分和复合主键相关，而在第三范式的这个例子中，E-mail这一列和主键就没有一点关联了。

要解决这一问题，我们需要将E-mail地址放到Accounts表中。图A-4显示了如何拆分Bugs表。由于在Accounts表中的E-mail是直接和主键关联而没有冗余的，因而这才是E-mail的合适位置。



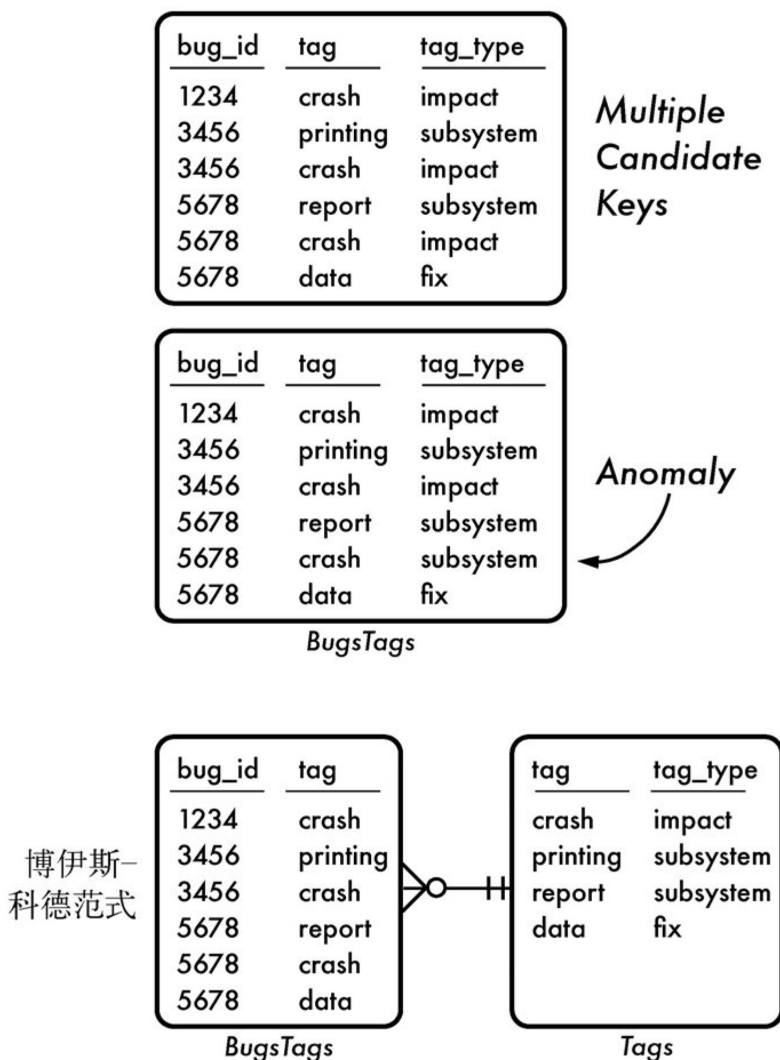
图**A-4**　冗余与第三范式

A.3.4　博伊斯—科德范式

比第三范式稍微严格一点的范式版本称为博伊斯—科德范式。这两个范式之间的不同之处在于，在第三范式中，所有的非关键字列都必须直接依赖于这张表中的关键字列，而在博伊斯—科德范式中，所有关键字列也必须遵循这一规则，这一点在一张表有多种列的集合可作为表的关键字时才有效。

比如，我们有三种Tag类型：描述Bug所造成影响的tag，描述Bug影响子系统的tag，以及Bug修复状态的tag。每一个Bug对于每一种Tag类型只能有一个tag。可能的键组合包括bug_id加上tag，或者bug_id加上tag_type。这两种组合都应该足够定位每一个独立的行。

在图A-5中，可以看到一个满足第三范式但是不满足博伊斯—科德范式的表，以及如何修改才能让它满足博伊斯—科德范式。



图**A-5**　第三范式与博伊斯—科德范式

A.3.5　第四范式

现在，我们需要修改数据库，以支持多个用户报告同一个Bug，并分配给多个开发工程师，然后由多个质量工程师验证。我们知道多对多

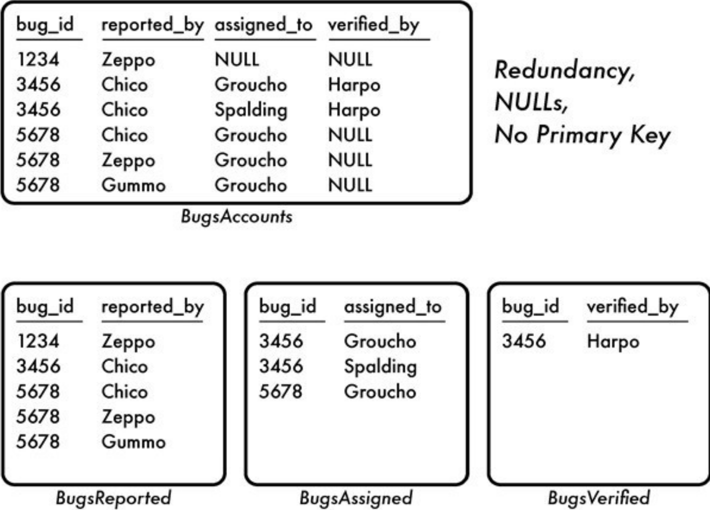
的关系需要一张额外的表：

Normalization/4NF-anti.sql

```
CREATE TABLE BugsAccounts (
  bug_id        BIGINT NOT NULL,
  reported_by   BIGINT,
  assigned_to   BIGINT,
  verified_by   BIGINT,
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id)
);
```

我们不能单独使用bug_id作为主键。每个Bug需要多行数据来实现各个字段都支持多个账号的目的。也不能将前两列或者前三列作为复合主键，因为这样，最后一列依旧不支持多个值。因此，主键必须是由所有四列组成。然而，assigned_to和verified_by需要可以为空，因为Bug在被指派和验证之前是可以报告的。而标准情况下，所有的主键列都有一个非空的约束。

另一问题就是当某一列的账号数小于其他列时，就会造成数据冗余。图A-6显示了这种冗余。



图**A-6**　合并关系与第四范式

上面提出的这些问题都是由于创建了一张做了双倍或者三倍工作的交叉表。当尝试使用一张交叉表描述多个多对多关系时，就会违背第四范式。

图A-6展示了我们可以拆分这张表来解决问题。我们应该为每一种多对多关系使用一张单独的交叉表，这就解决了冗余和数量不匹配的问题。

Normalization/4NF-normal.sql

```
CREATE TABLE BugsReported (
  bug_id        BIGINT NOT NULL,
  reported_by   BIGINT NOT NULL,
  PRIMARY KEY (bug_id, reported_by),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (reported_by) REFERENCES Accounts(account_id)
);
CREATE TABLE BugsAssigned (
  bug_id        BIGINT NOT NULL,
  assigned_to   BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id)
);

CREATE TABLE BugsVerified (
  bug_id        BIGINT NOT NULL,
  verified_by   BIGINT NOT NULL,
  PRIMARY KEY (bug_id, verified_by),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (verified_by) REFERENCES Accounts(account_id)
);
```

A.3.6　第五范式

任何满足博伊斯—科德范式并且没有复合主键的表将同时满足第五范式。我们可以通过下面的这个例子来更好地理解第五范式。

有些工程师只为几个产品服务。我们应该将数据库设计成让我们了解谁在为哪些产品服务，以及在修复哪些Bug，并且最小化数据的冗

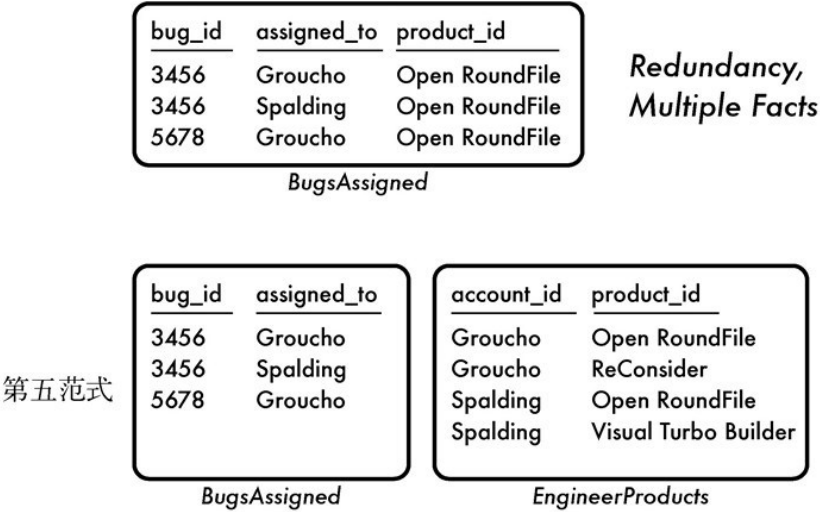
余。首先我们会想到在BugsAssigned表中增加一列来展示是哪个工程师在处理：

Normalization/5NF-anti.sql

```
CREATE TABLE BugsAssigned (
  bug_id          BIGINT NOT NULL,
  assigned_to     BIGINT NOT NULL,
  product_id      BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

但这不足以告诉我们这个工程师可以被指派为哪些产品服务，它只表明了这个工程师当前被指派去服务哪些产品，同时，这么做也造成了重复的存储。这是由于想在一张表中存储多种独立的多对多关系而产生的，就像我们在第四范式中所看到的问题一样。图A-7描述了这种冗余。[2]

2 图中使用了用户名字而不是ID。



图A-7 合并关系与第五范式

我们的解决方案是将每个关系都分别放到不同的表中：

Normalization/5NF-normal.sql

```
CREATE TABLE BugsAssigned (
  bug_id          BIGINT NOT NULL,
  assigned_to     BIGINT NOT NULL,
  PRIMARY KEY (bug_id, assigned_to),
  FOREIGN KEY (bug_id) REFERENCES Bugs(bug_id),
  FOREIGN KEY (assigned_to) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);

CREATE TABLE EngineerProducts (
  account_id      BIGINT NOT NULL,
  product_id      BIGINT NOT NULL,
  PRIMARY KEY (account_id, product_id),
  FOREIGN KEY (account_id) REFERENCES Accounts(account_id),
  FOREIGN KEY (product_id) REFERENCES Products(product_id)
);
```

现在我们可以记录某个工程师能为哪个产品服务，而不依赖于这个工程师是否正在为那个产品的Bug努力。

A.3.7　更多的范式

DK范式（Domain-Key normal form）认为表上的每个约束都是这张表的数据域约束和关键字约束的逻辑结果。DK范式涵盖了第三、四、五范式和博伊斯—科德范式。

比如说，一个状态为NEW或者DUPLICATE的Bug应该是没有任何工作量的，因此应该没有工作时间记录，也不需要在`verified_by`列中指派质量工程师。可能的实现方法是使用一个触发器或者一个`CHECK`约束。这些都是建在表的非关键字列上的约束，因而它们并不符合DK范式的标准。

第六范式旨在消除所有的联结依赖，它通常用来支持属性的变更历史。比如，我们可能想要在一张子表中记录下`Bugs.status`随着时间推移产生的变化：何时发生的变更，谁做的变更，以及其他可能的细节。

可以想象，如果让Bugs这张表满足第六范式，几乎每一列都需要附带一个历史记录表。这会导致表的数量过多。第六范式对于大多数程序来说都是没有必要的，但一些数据仓库技术里会使用到第六范式。3

> 3 比如，Anchor Modeling就是用了第六范式（http://www.anchormodeling.com）。

A.4 常识

规范化规则并不深奥或者复杂。它们只是减少冗余和提高数据一致性的惯用技术方法。

你可以使用这份关于关系和范式的简单参考资料，来帮助自己在未来的项目中更好地设计数据库。

附录B 参考书目

[BMMM98]	William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. *AntiPatterns*. John Wiley and Sons, Inc., New York, 1998.
[Cel04]	Joe Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann Publishers, San Francisco, 2004.
[Cel05]	Joe Celko. *Joe Celko's SQL Programming Style*. Morgan Kaufmann Publishers, San Francisco, 2005.
[Cod70]	Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.
[Eva03]	Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, Reading, MA, first edition, 2003.
[Fow03]	Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison Wesley Longman, Reading, MA, 2003.
[Gla92]	Robert L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, Reading, MA, 1992.
[Gol91]	David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, pages 5–48, March 1991. Reprinted http://www.validlab.com/goldberg/paper.pdf.

[GP03]	Peter Gulutzan and Trudy Pelzer. *SQL Performance Tuning*. Addison-Wesley, 2003.
[HLV05]	Michael Howard, David LeBlanc, and John Viega. *19 Deadly Sins of Software Security*. McGraw-Hill, Emeryville, California, 2005.
[HT00]	Andrew Hunt and David Thomas. *The Pragmatic Programmer:From Journeyman to Master*. Addison-Wesley, Reading, MA, 2000.
[Lar04]	Craig Larman. *Applying UML and Patterns: an Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall, Englewood Cliffs, NJ, third edition, 2004.
[RTH08]	Sam Ruby, David Thomas, and David Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, third edition, 2008.
[Spo02]	Joel Spolsky. The law of leaky abstractions. http://www.joelonsoftware.com/articles/LeakyAbstractions.html, 2002.
[SZT+08]	Baron Schwartz, Peter Zaitsev, Vadim Tkachenko, Jeremy Zawodny, Arjen Lentz, and Derek J. Balling. *High Performance MySQL*. O'Reilly Media, Inc., second edition, 2008.
[Tro06]	Vadim Tropashko. *SQL Design Patterns*. Rampant Techpress, Kittrell, NC, USA, 2006.

版权信息

书名：SQL必知必会（第4版）

作者：Ben Forta

译者：钟鸣， 刘晓霞

ISBN：978-7-115-31398-0

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

您购买的图灵电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

目录

引言
致谢
第1课 了解SQL
1.1 数据库基础
1.1.1 数据库
1.1.2 表
1.1.3 列和数据类型
1.1.4 行
1.1.5 主键
1.2 什么是SQL
1.3 动手实践
1.4 小结
第2课 检索数据
2.1 SELECT语句
2.2 检索单个列
2.3 检索多个列
2.4 检索所有列
2.5 检索不同的值
2.6 限制结果
2.7 使用注释
2.8 小结
第3课 排序检索数据
3.1 排序数据
3.2 按多个列排序
3.3 按列位置排序
3.4 指定排序方向
3.5 小结
第4课 过滤数据
4.1 使用WHERE子句
4.2 WHERE子句操作符
4.2.1 检查单个值
4.2.2 不匹配检查
4.2.3 范围值检查
4.2.4 空值检查
4.3 小结
第5课 高级数据过滤

5.1　组合WHERE子句
5.1.1　AND操作符
5.1.2　OR操作符
5.1.3　求值顺序
5.2　IN操作符
5.3　NOT操作符
5.4　小结
第6课　用通配符进行过滤
6.1　LIKE操作符
6.1.1　百分号（%）通配符
6.1.2　下划线（_）通配符
6.1.3　方括号（[ ]）通配符
6.2　使用通配符的技巧
6.3　小结
第7课　创建计算字段
7.1　计算字段
7.2　拼接字段
使用别名
7.3　执行算术计算
7.4　小结
第8课　使用函数处理数据
8.1　函数
函数带来的问题
8.2　使用函数
8.2.1　文本处理函数
8.2.2　日期和时间处理函数
8.2.3　数值处理函数
8.3　小结
第9课　汇总数据
9.1　聚集函数
9.1.1　AVG()函数
9.1.2　COUNT()函数
9.1.3　MAX()函数
9.1.4　MIN()函数
9.1.5　SUM()函数
9.2　聚集不同值
9.3　组合聚集函数
9.4　小结
第10课　分组数据
10.1　数据分组

10.2　创建分组
10.3　过滤分组
10.4　分组和排序
10.5　SELECT子句顺序
10.6　小结
第11课　使用子查询
11.1　子查询
11.2　利用子查询进行过滤
11.3　作为计算字段使用子查询
11.4　小结
第12课　联结表
12.1　联结
12.1.1　关系表
12.1.2　为什么使用联结
12.2　创建联结
12.2.1　WHERE子句的重要性
12.2.2　内联结
12.2.3　联结多个表
12.3　小结
第13课　创建高级联结
13.1　使用表别名
13.2　使用不同类型的联结
13.2.1　自联结
13.2.2　自然联结
13.2.3　外联结
13.3　使用带聚集函数的联结
13.4　使用联结和联结条件
13.5　小结
第14课　组合查询
14.1　组合查询
14.2　创建组合查询
14.2.1　使用UNION
14.2.2　UNION规则
14.2.3　包含或取消重复的行
14.2.4　对组合查询结果排序
14.3　小结
第15课　插入数据
15.1　数据插入
15.1.1　插入完整的行
15.1.2　插入部分行

15.1.3 插入检索出的数据
15.2 从一个表复制到另一个表
15.3 小结
第16课 更新和删除数据
16.1 更新数据
16.2 删除数据
16.3 更新和删除的指导原则
16.4 小结
第17课 创建和操纵表
17.1 创建表
17.1.1 表创建基础
17.1.2 使用NULL值
17.1.3 指定默认值
17.2 更新表
17.3 删除表
17.4 重命名表
17.5 小结
第18课 使用视图
18.1 视图
18.1.1 为什么使用视图
18.1.2 视图的规则和限制
18.2 创建视图
18.2.1 利用视图简化复杂的联结
18.2.2 用视图重新格式化检索出的数据
18.2.3 用视图过滤不想要的数据
18.2.4 使用视图与计算字段
18.3 小结
第19课 使用存储过程
19.1 存储过程
19.2 为什么要使用存储过程
19.3 执行存储过程
19.4 创建存储过程
19.5 小结
第20课 管理事务处理
20.1 事务处理
20.2 控制事务处理
20.2.1 使用ROLLBACK
20.2.2 使用COMMIT
20.2.3 使用保留点
20.3 小结

第21课　使用游标
21.1　游标
21.2　使用游标
21.2.1　创建游标
21.2.2　使用游标
21.2.3　关闭游标
21.3　小结
第22课　高级SQL特性
22.1　约束
22.1.1　主键
22.1.2　外键
22.1.3　唯一约束
22.1.4　检查约束
22.2　索引
22.3　触发器
22.4　数据库安全
22.5　小结
附录A　样例表脚本
A.1　样例表
A.2　获得样例表
A.2.1　下载可供使用的数据文件
A.2.2　下载DBMS SQL脚本
附录B　流行的应用程序
B.1　使用Apache Open Office Base
B.2　使用Adobe ColdFusion
B.3　使用IBM DB2
B.4　使用MariaDB
B.5　使用Microsoft Access
B.6　使用Microsoft ASP
B.7　使用Microsoft ASP.NET
B.8　使用Microsoft Query
B.9　使用Microsoft SQL Server（包括Microsoft SQL Server Express）
B.10　使用MySQL
B.11　使用Oracle
B.12　使用Oracle Express
B.13　使用PHP
B.14　使用PostgreSQL
B.15　使用SQLite
B.16　配置ODBC数据源

附录C　SQL语句的语法
C.1　ALTER TABLE
C.2　COMMIT
C.3　CREATE INDEX
C.4　CREATE PROCEDURE
C.5　CREATE TABLE
C.6　CREATE VIEW
C.7　DELETE
C.8　DROP
C.9　INSERT
C.10　INSERT SELECT
C.11　ROLLBACK
C.12　SELECT
C.13　UPDATE
附录D　SQL数据类型
D.1　字符串数据类型
D.2　数值数据类型
D.3　日期和时间数据类型
D.4　二进制数据类型
附录E　SQL保留字
常用SQL语句速查

引言

SQL是使用最为广泛的数据库语言。不管你是应用开发者、数据库管理员、Web应用设计师、移动应用开发人员，还是只使用Microsoft Office，掌握良好的SQL知识对用好数据库都是很重要的。

本书可以说是应需而生。我讲授了多年的Web应用开发，学生们经常要求我推荐一些SQL图书。SQL方面的书很多，有的其实很不错，但它们都有一个共同的特点，就是讲授的内容太多了，多数人其实不需要了解那么多。很多图书讲的不是SQL本身，而是从数据库设计、规范化到关系数据库理论以及管理问题等，事无巨细都讲一通。当然，这些内容也很重要，但大多数读者仅想学习SQL，他们未必感兴趣。

因此，我找不到合适书籍推荐给学生，只好把在课堂上给学生讲授的SQL知识汇编成了本书。本书将讲授读者需要了解的SQL知识，从简单的数据检索入手，逐步过渡到一些较为复杂的内容，如联结、子查询、存储过程、游标、触发器以及表约束等。读者将从本书中循序渐进、系统而直接地学到SQL的知识和技巧。

本书写到了第4版，它已经教会了英语国家近30万的读者使用SQL，并且还翻译出版了十多种其他语言的版本。现在轮到你了，让我们翻到第1课，开始学习吧。你将很快编写出世界级的SQL。

读者对象

本书适合以下读者：

- SQL新手；
- 希望快速学会并熟练使用SQL；
- 希望知道如何使用SQL开发应用程序；
- 希望在无人帮助的情况下有效而快速地使用SQL。

本书涵盖的DBMS

一般来说，本书中所讲授的SQL可以应用到任何数据库管理系统（DBMS）。但是，各种SQL实现不尽相同，本书介绍的SQL主要适用于以下系统（需要时会给出特殊说明和注释）：

- Apache Open Office Base；
- IBM DB2；
- Microsoft Access；
- Microsoft SQL Server（包括Microsoft SQL Server Express）；
- MariaDB；
- MySQL；
- Oracle（包括Oracle Express）；
- PostgreSQL；
- SQLite。

本书中的所有数据库示例（或者创建数据库示例的SQL脚本例子）对于这些DBMS都是适用的，它们可以在本书的网页http://forta.com/books/0672336073/上获得。

本书约定

本书采用等宽字体表示代码，读者输入的文本与应该出现在屏幕上的文本也都以等宽字体给出。如：

```
It will look like this to mimic the way text looks on your sc
```

变量和表达式的占位符用斜体表示，你可以用具体的值代替它。

代码行前的箭头（➥）表示代码太长，上一行容纳不下。在➥符号后输入的所有字符都应该是前一行的内容。

> 说明
> 给出上下文讨论中比较重要的信息。

> 提示
> 就某任务给出建议或更简单的方法。

> 注意
> 提醒可能出现的问题，避免出现事故。

> 新术语
> 清晰定义重要的新词汇。

输入▼
读者可以自己输入的代码，通常紧挨着代码出现。

输出▼
强调某个程序执行时的输出，通常出现在代码后。

分析▼
对程序代码进行逐行分析。

致谢

感谢Sams出版团队这些年来对我的支持、奉献和鼓励。特别要感谢Mark Taber，他促成了这本早就该更新的书，建议并帮助对书中的代码进行着色，更增加了新版的可读性，提升了书的价值。

感谢我的同事Greg Wilson，他为本书做了全面的技术审校。

感谢众多的读者对本书前三版提供的反馈。幸运的是，这些反馈多半是给与了我充分的肯定，在此深表感谢！这使得本版做了相应的改进和提高。欢迎大家对新版继续提出宝贵意见。

有好几十所学校将本书作为其IT和计算机科学课程的教材或参考书，在此特别表示感谢。能如此得到这些教授和老师的信任，对我是极大的鼓励，也让我诚惶诚恐。

最后，要感谢购买了本书前几版的接近30万的广大读者，你们使本书不仅成为我自己最畅销的一本书，而且也成为SQL方面最畅销的书之一。你们持续的支持是作者能得到的最宝贵的奖赏。

——Ben Forta

第1课　了解SQL

这一课介绍SQL究竟是什么，它能做什么事情。

1.1　数据库基础

你正在读这本SQL图书，这表明你需要以某种方式与数据库打交道。SQL正是用来实现这一任务的语言，因此在学习SQL之前，你应该对数据库及数据库技术的某些基本概念有所了解。

你可能还没有意识到，其实自己一直在使用数据库。每当你从电子邮件地址簿里查找名字时，就是在使用数据库。你在网站上进行搜索，也是在使用数据库。你在工作中登录网络，也需要依靠数据库验证用户名和密码。即使是在自动取款机上使用ATM卡，也要利用数据库进行密码验证和查询余额。

虽然我们一直都在使用数据库，但对究竟什么是数据库并不十分清楚。更何况人们可能会使用相同的数据库术语表示不同的事物，进一步加剧了这种混乱。因此，我们首先给出一些最重要的数据库术语，并加以说明。

> 提示：基本概念回顾
> 后面是一些基本数据库概念的简要介绍。如果你已经具有一定的数据库经验，可以借此复习巩固一下；如果你刚开始接触数据库，可以由此了解必需的基本知识。理解数据库概念是掌握SQL的重要前提，如果有必要，你或许还应该参阅其他一些有关数据库基础知识的书籍 。

1.1.1　数据库

数据库这个术语的用法很多，但就本书而言（从SQL的角度来看），数据库是一个以某种有组织的方式存储的数据集合。最简单的办法是将数据库想象为一个文件柜。这个文件柜是一个存放数据的物理位置，不管数据是什么，也不管数据是如何组织的。

> 数据库（**database**）
> 保存有组织的数据的容器（通常是一个文件或一组文件）。

> 注意：误用导致混淆
> 人们通常用数据库这个术语来代表他们使用的数据库软件，这是不正确的，也因此产生了许多混淆。确切地说，数据库软件应称为数据库管理系统（即DBMS）。数据库是通过DBMS创建和操纵的容器，而具体它究竟是什么，形式如何，各种数据库都不一样。

1.1.2　表

你往文件柜里放资料时，并不是随便将它们扔进某个抽屉就完事了，而是在文件柜中创建文件，然后将相关的资料放入特定的文件中。

在数据库领域中，这种文件称为表。表是一种结构化的文件，可用来存储某种特定类型的数据。表可以保存顾客清单、产品目录，或者其他信息清单。

> 表（**table**）
> 某种特定类型数据的结构化清单。

这里的关键一点在于，存储在表中的数据是同一种类型的数据或清单。决不应该将顾客的清单与订单的清单存储在同一个数据库表中，否则以后的检索和访问会很困难。应该创建两个表，每个清单一个表。

数据库中的每个表都有一个名字来标识自己。这个名字是唯一的，即数据库中没有其他表具有相同的名字。

> 说明：表名
> 使表名成为唯一的，实际上是数据库名和表名等的组合。有的数据库还使用数据库拥有者的名字作为唯一名的一部分。也就是说，虽然在相同数据库中不能两次使用相同的表名，但在不同的数据库中完全可以使用相同的表名。

表具有一些特性，这些特性定义了数据在表中如何存储，包含存储什么样的数据，数据如何分解，各部分信息如何命名等信息。描述表的这组信息就是所谓的模式（schema），模式可以用来描述数据库中特定的表，也可以用来描述整个数据库（和其中表的关系）。

> 模式
> 关于数据库和表的布局及特性的信息。

1.1.3 列和数据类型

表由列组成。列存储表中某部分的信息。

> 列（**column**）
> 表中的一个字段。所有表都是由一个或多个列组成的。

理解列的最好办法是将数据库表想象为一个网格，就像个电子表格那样。网格中每一列存储着某种特定的信息。例如，在顾客表中，一列存储顾客编号，另一列存储顾客姓名，而地址、城市、州以及邮政编码全都存储在各自的列中。

> 提示：分解数据
> 正确地将数据分解为多个列极为重要。例如，城市、州、邮政编码应该总是彼此独立的列。通过分解这些数据，才有可能利用特定的列对数据进行分类和过滤（如找出特定州或特定城市的所有顾客）。如果城市和州组合在一个列中，则按州进行分类或过滤就会很困难。
>
> 你可以根据自己的具体需求来决定把数据分解到何种程度。例如，一般可以把门牌号和街道名一起存储在地址里。这没有问题，除非你哪天想用街道名来排序，这时，最好将门牌号和街道名分开。

数据库中每个列都有相应的数据类型。数据类型（datatype）定义了列可以存储哪些数据种类。例如，如果列中存储的是数字（或许是订单中的物品数），则相应的数据类型应该为数值类型。如果列中存储的是日期、文本、注释、金额等，则应该规定好恰当的数据类型。

> 数据类型
> 所允许的数据的类型。每个表列都有相应的数据类型，它限制（或允许）该列中存储的数据。

数据类型限定了可存储在列中的数据种类（例如，防止在数值字段中录入字符值）。数据类型还帮助正确地分类数据，并在优化磁盘使用方面起重要的作用。因此，在创建表时必须特别关注所用的数据类

型。

> 注意：数据类型兼容
> 数据类型及其名称是SQL不兼容的一个主要原因。虽然大多数基本数据类型得到了一致的支持，但许多高级的数据类型却没有。更糟的是，偶然会有相同的数据类型在不同的DBMS中具有不同的名称。对此用户毫无办法，重要的是在创建表结构时要记住这些差异。

1.1.4　行

表中的数据是按行存储的，所保存的每个记录存储在自己的行内。如果将表想象为网格，网格中垂直的列为表列，水平行为表行。

例如，顾客表可以每行存储一个顾客。表中的行编号为记录的编号。

> 行（**row**）
> 表中的一个记录。

> 说明：是记录还是行？
> 你可能听到用户在提到行时称其为数据库记录（record）。这两个术语多半是可以交替使用的，但从技术上说，行才是正确的术语。

1.1.5　主键

表中每一行都应该有一列（或几列）可以唯一标识自己。顾客表可以使用顾客编号，而订单表可以使用订单ID。雇员表可以使用雇员ID或雇员社会安全号。

> 主键（**primary key**）
> 一列（或一组列），其值能够唯一标识表中每一行。

唯一标识表中每行的这个列（或这几列）称为主键。主键用来表示一个特定的行。没有主键，更新或删除表中特定行就极为困难，因为你不能保证操作只涉及相关的行。

> 提示：应该总是定义主键
> 虽然并不总是需要主键，但多数数据库设计者都会保证他们创建的每个表具有一个主键，以便于以后的数据操作和管理。

表中的任何列都可以作为主键，只要它满足以下条件：

- 任意两行都不具有相同的主键值；
- 每一行都必须具有一个主键值（主键列不允许NULL值）；
- 主键列中的值不允许修改或更新；
- 主键值不能重用（如果某行从表中删除，它的主键不能赋给以后的新行）。

主键通常定义在表的一列上，但并不是必需这么做，也可以一起使用多个列作为主键。在使用多列作为主键时，上述条件必须应用到所有列，所有列值的组合必须是唯一的（但单个列的值可以不唯一）。

还有一种非常重要的键，称为外键，我们将在第12课中介绍。

1.2　什么是SQL

SQL（发音为字母S-Q-L或sequel）是结构化查询语言（Structured Query Language）的缩写。SQL是一种专门用来与数据库沟通的语言。

与其他语言（如英语或Java、C、PHP这样的编程语言）不一样，SQL中只有很少的词，这是有意而为的。设计SQL的目的是很好地完成一项任务——提供一种从数据库中读写数据的简单有效的方法。

SQL有如下的优点。

- SQL不是某个特定数据库供应商专有的语言。几乎所有重要的DBMS都支持SQL，所以学习此语言使你几乎能与所有数据库打交道。
- SQL简单易学。它的语句全都是由有很强描述性的英语单词组成，而且这些单词的数目不多。
- SQL虽然看上去很简单，但实际上是一种强有力的语言，灵活使用其语言元素，可以进行非常复杂和高级的数据库操作。

下面我们将开始真正学习SQL。

> 说明：**SQL**的扩展
> 许多DBMS厂商通过增加语句或指令，对SQL进行了扩展。这种扩展的目的是提供执行特定操作的额外功能或简化方法。虽然这种扩展很有用，但一般都是针对个别DBMS的，很少有两个以上的供应商支持这种扩展。
>
> 标准SQL由ANSI标准委员会管理，从而称为ANSI SQL。所有主要的DBMS，即使有自己的扩展，也都支持ANSI SQL。各个实现有自己的名称，如PL/SQL、Transact-SQL 等。
>
> 本书讲授的SQL主要是ANSI SQL。在使用某种DBMS特定的SQL时，会特别说明。

1.3　动手实践

与其他任何语言一样，学习SQL的最好方法是自己动手实践。为此，需要一个数据库和用来测试SQL语句的应用系统。

本书中所有课程采用的都是真实的SQL语句和数据表。附录A给出了具体的样例表，并介绍了获得（或创建）它们的详细步骤，便于读者理解每一课讲授的内容。附录B介绍在各种应用程序中执行SQL所需的步骤。在进入下一课之前，强烈建议读者先阅读这两个附录的内容，为以后的学习做好准备。

1.4　小结

这一课介绍了什么是SQL，它为什么很有用。因为SQL是用来与数据库打交道的，所以，我们也复习了一些基本的数据库术语。

第2课　检索数据

这一课介绍如何使用SELECT语句从表中检索一个或多个数据列。

2.1　SELECT语句

正如第1课所述，SQL语句是由简单的英语单词构成的。这些单词称为关键字，每个SQL语句都是由一个或多个关键字构成的。最经常使用的SQL语句大概就是SELECT语句了。它的用途是从一个或多个表中检索信息。

> 关键字（**keyword**）
> 作为SQL组成部分的保留字。关键字不能用作表或列的名字。附录E列出了某些经常使用的保留字。

为了使用SELECT检索表数据，必须至少给出两条信息——想选择什么，以及从什么地方选择。

> 说明：理解例子
> 本书各课程中的样例SQL语句（和样例输出）使用了附录A中描述的一组数据文件。如果想要理解和试验这些样例（我强烈建议这样做），请参阅附录A，它解释了如何下载或创建这些数据文件。
>
> 重要的是，要理解SQL是一种语言而不是一个应用程序。具体如何写SQL语句并显示语句输出，是随不同的应用程序而变化的。为帮助读者根据自己的环境使用相应的例子，附录B介绍了如何针对许多流行的应用程序及开发环境发出本书中介绍的语句。如果读者需要了解某个应用程序，附录B中也给出了相应的建议。

2.2　检索单个列

我们将从简单的SQL SELECT语句讲起，此语句如下所示：

输入▼

```
SELECT prod_name
```

```
FROM Products;
```

分析▼

上述语句利用SELECT语句从Products表中检索一个名为prod_name的列。所需的列名写在SELECT关键字之后，FROM关键字指出从哪个表中检索数据。此语句的输出如下所示：

输出▼

```
prod_name
-------------------
Fish bean bag toy
Bird bean bag toy
Rabbit bean bag toy
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
Raggedy Ann
King doll
Queen doll
```

> 提示：未排序数据
> 如果你自己试验这个查询，可能会发现显示输出的数据顺序与这里的不同。出现这种情况很正常。如果没有明确排序查询结果（下一课介绍），则返回的数据没有特定的顺序。返回数据的顺序可能是数据被添加到表中的顺序，也可能不是。只要返回相同数目的行，就是正常的。

如上的一条简单SELECT语句将返回表中的所有行。数据没有过滤（过滤将得出结果集的一个子集），也没有排序。以后几课将讨论这些内容。

> 提示：结束**SQL**语句
> 多条SQL语句必须以分号（；）分隔。多数DBMS不需要在单条SQL语句后加分号，但也有DBMS可能必须在单条SQL语句后加上分号。当然，如果愿意可以总是加上分号。事实上，即使不一定需要，加上分号也肯定没有坏处。

> 提示：**SQL语句和大小写**
> 请注意，SQL语句不区分大小写，因此`SELECT`与`select`是相同的。同样，写成`Select`也没有关系。许多SQL开发人员喜欢对SQL关键字使用大写，而对列名和表名使用小写，这样做使代码更易于阅读和调试。不过，一定要认识到虽然SQL是不区分大小写的，但是表名、列名和值可能有所不同（这有赖于具体的DBMS及其如何配置）。

> 提示：使用空格
> 在处理SQL语句时，其中所有空格都被忽略。SQL语句可以写成长长的一行，也可以分写在多行。下面这三种写法的作用是一样的。
>
> ```
> SELECT prod_name
> FROM Products;
>
> SELECT prod_name FROM Products;
>
> SELECT
> prod_name
> FROM
> Products;
> ```
>
> 多数SQL开发人员认为，将SQL语句分成多行更容易阅读和调试。

2.3 检索多个列

要想从一个表中检索多个列，仍然使用相同的`SELECT`语句。唯一的不同是必须在`SELECT`关键字后给出多个列名，列名之间必须以逗号分隔。

> 提示：当心逗号
> 在选择多个列时，一定要在列名之间加上逗号，但最后一个列名后不加。如果在最后一个列名后加了逗号，将出现错误。

下面的`SELECT`语句从`Products`表中选择3列。

输入▼

```
SELECT prod_id, prod_name, prod_price
FROM Products;
```

分析▼
与前一个例子一样，这条语句使用SELECT语句从表Products中选择数据。在这个例子中，指定了3个列名，列名之间用逗号分隔。此语句的输出如下：

输出▼

```
prod_id          prod_name                 prod_price
---------        --------------------      ----------
BNBG01           Fish bean bag toy         3.4900
BNBG02           Bird bean bag toy         3.4900
BNBG03           Rabbit bean bag toy       3.4900
BR01             8 inch teddy bear         5.9900
BR02             12 inch teddy bear        8.9900
BR03             18 inch teddy bear        11.9900
RGAN01           Raggedy Ann               4.9900
RYL01            King doll                 9.4900
RYL02            Queen dool                9.4900
```

说明：数据表示
从上述输出可以看到，SQL语句一般返回原始的、无格式的数据。数据的格式化是表示问题，而不是检索问题。因此，表示（如把上面的价格值显示为正确的十进制数值货币金额）一般在显示该数据的应用程序中规定。通常很少直接使用实际检索出的数据（没有应用程序提供的格式）。

2.4　检索所有列

除了指定所需的列外（如上所述，一个或多个列），SELECT语句还可以检索所有的列而不必逐个列出它们。在实际列名的位置使用星号（*）通配符可以做到这点，如下所示。

输入▼

```
SELECT *
```

```
FROM Products;
```

分析▼

如果给定一个通配符（*），则返回表中所有列。列的顺序一般是列在表定义中出现的物理顺序，但并不总是如此。不过，SQL数据很少这样（通常，数据返回给应用程序，根据需要进行格式化，再表示出来）。因此，这不应该造成什么问题。

> 警告：使用通配符
> 一般而言，除非你确实需要表中的每一列，否则最好别使用*通配符。虽然使用通配符能让你自己省事，不用明确列出所需列，但检索不需要的列通常会降低检索和应用程序的性能。

> 提示：检索未知列
> 使用通配符有一个大优点。由于不明确指定列名（因为星号检索每一列），所以能检索出名字未知的列。

2.5　检索不同的值

如前所述，SELECT语句返回所有匹配的行。但是，如果你不希望每个值每次都出现，该怎么办呢？例如，你想检索products表中所有产品供应商的ID：

输入▼

```
SELECT vend_id
FROM Products;
```

输出▼

```
vend_id
----------
BRS01
BRS01
BRS01
DLL01
DLL01
DLL01
```

```
DLL01
FNG01
FNG01
```

SELECT语句返回9行（即使表中只有3个产品供应商），因为products表中有9种产品。那么如何检索出不同的值？

办法就是使用DISTINCT关键字，顾名思义，它指示数据库只返回不同的值。

输入▼

```
SELECT DISTINCT vend_id
FROM Products;
```

分析▼ SELECT DISTINCT vend_id告诉DBMS只返回不同（具有唯一性）的vend_id行，所以正如下面的输出，只有3行。如果使用DISTINCT关键字，它必须直接放在列名的前面。

输出▼

```
vend_id
----------
BRS01
DLL01
FNG01
```

> 警告：不能部分使用**DISTINCT**
> DISTINCT关键字作用于所有的列，不仅仅是跟在其后的那一列。例如，你指定SELECT DISTINCT vend_id, prod_price，因为指定的两列不完全相同，所以所有的行都会被检索出来。

2.6　限制结果

SELECT语句返回指定表中所有匹配的行，很可能是每一行。如果你只想返回第一行或者一定数量的行，该怎么办呢？这是可行的，然而遗憾的是，各种数据库中的这一SQL实现并不相同。

在SQL Server和Access中使用SELECT时，可以使用TOP关键字来限制

最多返回多少行，如下所示：

输入▼

```
SELECT TOP 5 prod_name
FROM Products;
```

输出▼

```
prod_name
-----------------
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
Fish bean bag toy
Bird bean bag toy
```

分析▼

上面代码使用SELECT TOP 5语句，只检索前5行数据。

如果你使用的是DB2，很可能习惯使用下面这一DBMS特定的SQL语句，像这样：

输入▼

```
SELECT prod_name
FROM Products
FETCH FIRST 5 ROWS ONLY;
```

分析▼

FETCH FIRST 5 ROWS ONLY就会按字面的意思去做的。

如果你使用Oracle，需要基于ROWNUM（行计数器）来计算行，像这样：

输入▼

```
SELECT prod_name
FROM Products
WHERE ROWNUM <=5;
```

如果你使用MySQL、MariaDB、PostgreSQL或者SQLite，需要使用 LIMIT 子句，像这样：

输入▼

```
SELECT prod_name
FROM Products
LIMIT 5;
```

分析▼

上述代码使用SELECT语句来检索单独的一列数据。LIMIT 5指示MySQL等DBMS返回不超过5行的数据。这个语句的输出参见下面的代码。

为了得到后面的5行数据，需要指定从哪儿开始以及检索的行数，像这样：

输入▼

```
SELECT prod_name
FROM Products
LIMIT 5 OFFSET 5;
```

分析▼

LIMIT 5 OFFSET 5指示MySQL等DBMS返回从第5行起的5行数据。第一个数字是检索的行数，第二个数字是指从哪儿开始。这个语句的输出是：

输出▼

```
prod_name
-------------------
Rabbit bean bag toy
Raggedy Ann
King doll
Queen doll
```

所以，LIMIT指定返回的行数。LIMIT 所带的 OFFSET指定从哪儿开始。在我们的例子中，Products表中只有9种产品，所以LIMIT 5 OFFSET 5只返回了4行数据（因为没有第5行）。

> 警告：第**0**行
> 第一个被检索的行是第0行，而不是第1行。因此，`LIMIT 1 OFFSET 1`会检索第2行，而不是第1行。

> 提示：**MySQL**、 **MariaDB**和**SQLite**捷径
> MySQL、 MariaDB和SQLite支持简化版的`LIMIT 4 OFFSET 3`语句，即`LIMIT 3,4`。使用这个语法，逗号之前的值对应`OFFSET`， 逗号之后的值对应`LIMIT`。

> 说明：并非所有的**SQL**实现都一样
> 我加入这一节只有一个原因，就是要说明，SQL虽然通常都有相当一致的实现，但你不能想当然地认为它总是这样。非常基本的语句往往是容易移植的，但较复杂的语句就不同了。当你针对某个问题寻找SQL解决方案时，一定要记住这一点。

2.7　使用注释

可以看到，SQL语句是由DBMS处理的指令。如果你希望包括不进行处理和执行的文本，该怎么办呢？为什么你想要这么做呢？原因有以下几点。

- 我们这里使用的SQL语句都很短，也很简单。然而，随着你的SQL语句变长，复杂性增加，你就会想添加一些描述性的注释，这便于你自己今后参考，或者供项目后续参与人员参考。这些注释需要嵌入在SQL脚本中，但显然不能进行实际的DBMS处理。（相关示例可以参见附录B中使用的create.sql和populate.sql。）
- 这同样适用于SQL文件开始处的内容，它可能包含程序员的联系方式、程序描述以及一些说明。（相关示例也可参见附录B中的那些.sql文件。）
- 注释的另一个重要应用是暂时停止要执行的SQL代码。如果你碰到一个长SQL语句，而只想测试它的一部分，那么应该注释掉一些代码，以便DBMS将其视为注释而加以忽略。

很多DBMS都支持各种形式的注释语法。我们先来看行内注释：

输入▼

```
SELECT prod_name     -- 这是一条注释
FROM Products;
```

分析▼

注释使用--（两个连字符）嵌在行内。-- 之后的文本就是注释，例如，这用来描述CREATE TABLE语句中的列就很不错。

下面是另一种形式的行内注释（虽然这种形式很少得到支持）。

输入▼

```
# 这是一条注释
SELECT prod_name
FROM Products;
```

分析▼

在一行的开始处使用#，这一整行都将作为注释。你在本书提供的脚本create.sql和populate.sql中可以看到这种形式的注释。

你也可以进行多行注释，注释可以在脚本的任何位置停止和开始。

输入▼

```
/* SELECT prod_name, vend_id
FROM Products; */
SELECT prod_name
FROM Products;
```

分析▼

注释从/*开始，到*/结束，/*和*/之间的任何内容都是注释。这种方式常用于给代码加注释，就如这个例子演示的，这里定义了两个SELECT语句，但是第一个不会执行，因为它已经被注释掉了。

2.8　小结

这一课学习了如何使用SQL的`SELECT`语句来检索单个表列、多个表列以及所有表列。你也学习了如何返回不同的值，如何注释代码。同时不幸的是，更复杂的SQL使得SQL代码变得不轻便。下一课将讲授如何对检索出来的数据进行排序。

第3课　排序检索数据

这一课讲授如何使用SELECT语句的ORDER BY子句，根据需要排序检索出的数据。

3.1　排序数据

正如上一课所述，下面的SQL语句返回某个数据库表的单个列。但请看其输出，并没有特定的顺序。

输入▼

```
SELECT prod_name
FROM Products;
```

输出▼

```
prod_name
--------------------
Fish bean bag toy
Bird bean bag toy
Rabbit bean bag toy
8 inch teddy bear
12 inch teddy bear
18 inch teddy bear
Raggedy Ann
King doll
Queen doll
```

其实，检索出的数据并不是随机显示的。如果不排序，数据一般将以它在底层表中出现的顺序显示，这有可能是数据最初添加到表中的顺序。但是，如果数据随后进行过更新或删除，那么这个顺序将会受到DBMS重用回收存储空间的方式的影响。因此，如果不明确控制的话，则最终的结果不能（也不应该）依赖该排序顺序。关系数据库设计理论认为，如果不明确规定排序顺序，则不应该假定检索出的数据的顺序有任何意义。

> **子句（clause）**

> SQL语句由子句构成，有些子句是必需的，有些则是可选的。一个子句通常由一个关键字加上所提供的数据组成。子句的例子有我们在前一课看到的SELECT语句的FROM子句。

为了明确地排序用SELECT语句检索出的数据，可使用ORDER BY子句。ORDER BY子句取一个或多个列的名字，据此对输出进行排序。请看下面的例子：

输入▼

```
SELECT prod_name
FROM Products
ORDER BY prod_name;
```

分析▼
除了指示DBMS软件对prod_name列以字母顺序排序数据的ORDER BY子句外，这条语句与前面的语句相同。结果如下。

输出▼

```
prod_name
--------------------
12 inch teddy bear
18 inch teddy bear
8 inch teddy bear
Bird bean bag toy
Fish bean bag toy
King doll
Queen doll
Rabbit bean bag toy
Raggedy Ann
```

> **ORDER BY**子句的位置
> 在指定一条ORDER BY子句时，应该保证它是SELECT语句中最后一条子句。如果它不是最后的子句，将会出现错误消息。

> 通过非选择列进行排序
> 通常，ORDER BY子句中使用的列将是为显示而选择的列。但是，实际上并不一定要这样，用非检索的列排序数据是完全合法

的。

3.2　按多个列排序

经常需要按不只一个列进行数据排序。例如，如果要显示雇员名单，可能希望按姓和名排序（首先按姓排序，然后在每个姓中再按名排序）。如果多个雇员有相同的姓，这样做很有用。

要按多个列排序，简单指定列名，列名之间用逗号分开即可（就像选择多个列时那样）。

下面的代码检索3个列，并按其中两个列对结果进行排序——首先按价格，然后按名称排序。

输入▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price, prod_name;
```

输出▼

```
prod_id     prod_price     prod_name
-------     ----------     --------------------
BNBG02      3.4900         Bird bean bag toy
BNBG01      3.4900         Fish bean bag toy
BNBG03      3.4900         Rabbit bean bag toy
RGAN01      4.9900         Raggedy Ann
BR01        5.9900         8 inch teddy bear
BR02        8.9900         12 inch teddy bear
RYL01       9.4900         King doll
RYL02       9.4900         Queen doll
BR03        11.9900        18 inch teddy bear
```

重要的是理解在按多个列排序时，排序的顺序完全按规定进行。换句话说，对于上述例子中的输出，仅在多个行具有相同的prod_price值时才对产品按prod_name进行排序。如果prod_price列中所有的值都是唯一的，则不会按prod_name排序。

3.3　按列位置排序

除了能用列名指出排序顺序外，ORDER BY还支持按相对列位置进行排序。为理解这一内容，我们来看个例子：

输入▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY 2, 3;
```

输出▼

```
prod_id     prod_price     prod_name
-------     ----------     --------------------
BNBG02      3.4900         Bird bean bag toy
BNBG01      3.4900         Fish bean bag toy
BNBG03      3.4900         Rabbit bean bag toy
RGAN01      4.9900         Raggedy Ann
BR01        5.9900         8 inch teddy bear
BR02        8.9900         12 inch teddy bear
RYL01       9.4900         King doll
RYL02       9.4900         Queen doll
BR03        11.9900        18 inch teddy bear
```

分析▼

可以看到，这里的输出与上面的查询相同，不同之处在于ORDER BY子句。SELECT清单中指定的是选择列的相对位置而不是列名。ORDER BY 2表示按SELECT清单中的第二个列prod_price进行排序。ORDER BY 2，3表示先按prod_price，再按prod_name进行排序。

这一技术的主要好处在于不用重新输入列名。但它也有缺点。首先，不明确给出列名有可能造成错用列名排序。其次，在对SELECT清单进行更改时容易错误地对数据进行排序（忘记对ORDER BY子句做相应的改动）。最后，如果进行排序的列不在SELECT清单中，显然不能使用这项技术。

> 提示：按非选择列排序
> 显然，当根据不出现在SELECT清单中的列进行排序时，不能采用这项技术。但是，如果有必要，可以混合使用实际列名和相对列位置。

3.4 指定排序方向

数据排序不限于升序排序（从A到Z），这只是默认的排序顺序。还可以使用ORDER BY子句进行降序（从Z到A）排序。为了进行降序排序，必须指定DESC关键字。

下面的例子以价格降序来排序产品（最贵的排在最前面）：

输入▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price DESC;
```

输出▼

```
prod_id      prod_price      prod_name
-------      ----------      --------------------
BR03         11.9900         18 inch teddy bear
RYL01        9.4900          King doll
RYL02        9.4900          Queen doll
BR02         8.9900          12 inch teddy bear
BR01         5.9900          8 inch teddy bear
RGAN01       4.9900          Raggedy Ann
BNBG01       3.4900          Fish bean bag toy
BNBG02       3.4900          Bird bean bag toy
BNBG03       3.4900          Rabbit bean bag toy
```

如果打算用多个列排序，该怎么办？下面的例子以降序排序产品（最贵的在最前面），再加上产品名：

输入▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
ORDER BY prod_price DESC, prod_name;
```

输出▼

```
prod_id      prod_price      prod_name
-------      ----------      --------------------
```

```
BR03          11.9900         18 inch teddy bear
RYL01         9.4900          King doll
RYL02         9.4900          Queen doll
BR02          8.9900          12 inch teddy bear
BR01          5.9900          8 inch teddy bear
RGAN01        4.9900          Raggedy Ann
BNBG02        3.4900          Bird bean bag toy
BNBG01        3.4900          Fish bean bag toy
BNBG03        3.4900          Rabbit bean bag toy
```

分析▼

DESC关键字只应用到直接位于其前面的列名。在上例中，只对prod_price列指定DESC，对prod_name列不指定。因此，prod_price列以降序排序，而prod_name列（在每个价格内）仍然按标准的升序排序。

> 警告：在多个列上降序排序
> 如果想在多个列上进行降序排序，必须对每一列指定DESC关键字。

请注意，DESC是DESCENDING的缩写，这两个关键字都可以使用。与DESC相对的是ASC（或ASCENDING），在升序排序时可以指定它。但实际上，ASC没有多大用处，因为升序是默认的（如果既不指定ASC也不指定DESC，则假定为ASC）。

> 提示：区分大小写和排序顺序
> 在对文本性数据进行排序时，A与a相同吗？a位于B之前，还是Z之后？这些问题不是理论问题，其答案取决于数据库的设置方式。
>
> 在字典（dictionary）排序顺序中，A被视为与a相同，这是大多数数据库管理系统的默认行为。但是，许多DBMS允许数据库管理员在需要时改变这种行为（如果你的数据库包含大量外语字符，可能必须这样做）。
>
> 这里的关键问题是，如果确实需要改变这种排序顺序，用简单的ORDER BY子句可能做不到。你必须请求数据库管理员的帮助。

3.5　小结

这一课学习了如何用SELECT语句的ORDER BY子句对检索出的数据进行排序。这个子句必须是SELECT语句中的最后一条子句。根据需要，可以利用它在一个或多个列上对数据进行排序。

第4课　过滤数据

这一课将讲授如何使用SELECT语句的WHERE子句指定搜索条件。

4.1　使用WHERE子句

数据库表一般包含大量的数据，很少需要检索表中的所有行。通常只会根据特定操作或报告的需要提取表数据的子集。只检索所需数据需要指定搜索条件（search criteria），搜索条件也称为过滤条件（filter condition）。

在SELECT语句中，数据根据WHERE子句中指定的搜索条件进行过滤。WHERE子句在表名（FROM子句）之后给出，如下所示：

输入▼

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price = 3.49;
```

分析▼
这条语句从products表中检索两个列，但不返回所有行，只返回prod_price值为3.49的行，如下所示：

输出▼

```
prod_name                   prod_price
-------------------         ----------
Fish bean bag toy           3.49
Bird bean bag toy           3.49
Rabbit bean bag toy         3.49
```

这个示例使用了简单的相等检验：检查这一列的值是否为指定值，据此过滤数据。不过，SQL不只能测试等于，还能做更多的事情。

> 提示：有多少个0？
> 你在练习这个示例时，会发现显示的结果可能是3.49、3.490、3.4900等。出现这样的情况，往往是因为DBMS指定了所使用的

数据类型及其默认行为。所以，如果你的输出可能与书上的有点不同，不必焦虑，毕竟从数学角度讲，3.49和3.4900是一样的。

提示：**SQL过滤与应用过滤**
数据也可以在应用层过滤。为此，SQL的SELECT语句为客户端应用检索出超过实际所需的数据，然后客户端代码对返回数据进行循环，提取出需要的行。

通常，这种做法极其不妥。优化数据库后可以更快速有效地对数据进行过滤。而让客户端应用（或开发语言）处理数据库的工作将会极大地影响应用的性能，并且使所创建的应用完全不具备可伸缩性。此外，如果在客户端过滤数据，服务器不得不通过网络发送多余的数据，这将导致网络带宽的浪费。

警告：**WHERE**子句的位置
在同时使用ORDER BY和WHERE子句时，应该让ORDER BY位于WHERE之后，否则将会产生错误（关于ORDER BY的使用，请参阅第3课）。

4.2 WHERE子句操作符

我们在做相等检验时看到了第一个WHERE子句，它确定一个列是否包含指定的值。SQL支持表4-1列出的所有条件操作符。

表**4-1** **WHERE**子句操作符

操作符	说明
=	等于
< >	不等于
!=	不等于
<	小于
< =	小于等于
!<	不小于
>	大于
> =	大于等于
!>	不大于
BETWEEN	在指定的两个值之间
IS NULL	为NULL值

> 警告：操作符兼容
> 表4-1中列出的某些操作符是冗余的（如< >与!=相同，!<相当于>=）。并非所有DBMS都支持这些操作符。想确定你的DBMS支持哪些操作符，请参阅相应的文档。

4.2.1　检查单个值

我们已经看到了检验相等的例子，现在来看看几个使用其他操作符的例子。

第一个例子是列出所有价格小于10美元的产品：

输入▼

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price < 10;
```

输出▼

```
prod_name                prod_price
-------------------      ----------
Fish bean bag toy        3.49
Bird bean bag toy        3.49
Rabbit bean bag toy      3.49
8 inch teddy bear        5.99
12 inch teddy bear       8.99
Raggedy Ann              4.99
King doll                9.49
Queen doll               9.49
```

下一条语句检索所有价格小于等于10美元的产品（因为没有价格恰好是10美元的产品，所以结果与前一个例子相同）：

输入▼

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price <= 10;
```

4.2.2　不匹配检查

这个例子列出所有不是供应商DLL01制造的产品：

输入▼

```
SELECT vend_id, prod_name
FROM Products
WHERE vend_id <> 'DLL01';
```

输出▼

```
vend_id          prod_name
----------       ------------------
BRS01            8 inch teddy bear
BRS01            12 inch teddy bear
BRS01            18 inch teddy bear
FNG01            King doll
FNG01            Queen doll
```

> 提示：何时使用引号
> 如果仔细观察上述WHERE子句中的条件，会看到有的值括在单引号内，而有的值未括起来。单引号用来限定字符串。如果将值与字符串类型的列进行比较，就需要限定引号。用来与数值列进行比较的值不用引号。

下面是相同的例子，其中使用!=而不是<>操作符：

输入▼

```
SELECT vend_id, prod_name
FROM Products
WHERE vend_id != 'DLL01';
```

> 警告：是**!=**还是**<>**？
> !=和<>通常可以互换。但是，并非所有DBMS都支持这两种不等于操作符。例如，Microsoft Access支持<>而不支持!=。如果有疑问，请参阅相应的DBMS文档。

4.2.3　范围值检查

要检查某个范围的值，可以使用BETWEEN操作符。其语法与其他

WHERE子句的操作符稍有不同，因为它需要两个值，即范围的开始值和结束值。例如，BETWEEN操作符可用来检索价格在5美元和10美元之间的所有产品，或在指定的开始日期和结束日期之间的所有日期。

下面的例子说明如何使用BETWEEN操作符，它检索价格在5美元和10美元之间的所有产品：

输入▼

```
SELECT prod_name, prod_price
FROM Products
WHERE prod_price BETWEEN 5 AND 10;
```

输出▼

```
prod_name                   prod_price
-------------------         ----------
8 inch teddy bear           5.99
12 inch teddy bear          8.99
King doll                   9.49
Queen doll                  9.49
```

分析▼

从这个例子可以看到，在使用BETWEEN时，必须指定两个值——所需范围的低端值和高端值。这两个值必须用AND关键字分隔。BETWEEN匹配范围中所有的值，包括指定的开始值和结束值。

4.2.4 空值检查

在创建表时，表设计人员可以指定其中的列能否不包含值。在一个列不包含值时，称其包含空值NULL。

> **NULL**
> 无值（no value），它与字段包含0、空字符串或仅仅包含空格不同。

确定值是否为NULL，不能简单地检查是否= NULL。SELECT语句有一个特殊的WHERE子句，可用来检查具有NULL值的列。这个WHERE子句就是IS NULL子句。其语法如下：

输入▼

```
SELECT prod_name
FROM Products
WHERE prod_price IS NULL;
```

这条语句返回所有没有价格（空prod_price字段，不是价格为0）的产品，由于表中没有这样的行，所以没有返回数据。但是，Customers表确实包含具有NULL值的列：如果没有电子邮件地址，则cust_email列将包含NULL值：

输入▼

```
SELECT cust_name
FROM Customers
WHERE cust_email IS NULL;
```

输出▼

```
cust_name
----------
Kids Place
The Toy Store
```

> 提示：各**DBMS**特有的操作符
> 许多DBMS扩展了标准的操作符集，提供了更高级的过滤选择。更多信息请参阅相应的DBMS文档。
>
> 警告：**NULL**和非匹配
> 通过过滤选择不包含指定值的所有行时，你可能希望返回含NULL值的行。但是这做不到。因为未知（unknown）有特殊的含义，数据库不知道它们是否匹配，所以在进行匹配过滤或非匹配过滤时，不会返回这些结果。
>
> 过滤数据时，一定要验证被过滤列中含NULL的行确实出现在返回的数据中。

4.3 小结

这一课介绍了如何用SELECT语句的WHERE子句过滤返回的数据。我们学习了如何检验相等、不相等、大于、小于、值的范围以及NULL值等。

第5课　高级数据过滤

这一课讲授如何组合WHERE子句以建立功能更强、更高级的搜索条件。我们还将学习如何使用NOT和IN操作符。

5.1　组合WHERE子句

第4课介绍的所有WHERE子句在过滤数据时使用的都是单一的条件。为了进行更强的过滤控制，SQL允许给出多个WHERE子句。这些子句有两种使用方式，即以AND子句或OR子句的方式使用。

> 操作符（**operator**）
> 用来联结或改变WHERE子句中的子句的关键字，也称为逻辑操作符（logical operator）。

5.1.1　AND操作符

要通过不只一个列进行过滤，可以使用AND操作符给WHERE子句附加条件。下面的代码给出了一个例子：

输入▼

```
SELECT prod_id, prod_price, prod_name
FROM Products
WHERE vend_id = 'DLL01' AND prod_price <= 4;
```

分析▼
此SQL语句检索由供应商DLL01制造且价格小于等于4美元的所有产品的名称和价格。这条SELECT语句中的WHERE子句包含两个条件，用AND关键字联结在一起。AND指示DBMS只返回满足所有给定条件的行。如果某个产品由供应商DLL01制造，但价格高于4美元，则不检索它。类似地，如果产品价格小于4美元，但不是由指定供应商制造的也不被检索。这条SQL语句产生的输出如下：

输出▼

```
prod_id       prod_price       prod_name
```

```
-------      ----------      --------------------
BNBG02       3.4900          Bird bean bag toy
BNBG01       3.4900          Fish bean bag toy
BNBG03       3.4900          Rabbit bean bag toy
```

> **AND**
> 用在WHERE子句中的关键字，用来指示检索满足所有给定条件的行。

这个例子只包含一个AND子句，因此最多有两个过滤条件。可以增加多个过滤条件，每个条件间都要使用AND关键字。

> 说明：没有**ORDER BY**子句
> 为了节省空间，也为了减少你的输入，我在很多例子里省略了ORDER BY子句。因此，你的输出完全有可能与书上的输出不一致。虽然返回行的数量总是对的，但它们的顺序可能不同。当然，如果你愿意也可以加上一个ORDER BY子句，它应该放在WHERE子句之后。

5.1.2　OR操作符

OR操作符与AND操作符正好相反，它指示DBMS检索匹配任一条件的行。事实上，许多DBMS在OR WHERE子句的第一个条件得到满足的情况下，就不再计算第二个条件了（在第一个条件满足时，不管第二个条件是否满足，相应的行都将被检索出来）。

请看如下的SELECT语句：

输入▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = ‘BRS01’;
```

分析▼
此SQL语句检索由任一个指定供应商制造的所有产品的产品名和价格。OR操作符告诉DBMS匹配任一条件而不是同时匹配两个条件。如果这里使用的是AND操作符，则没有数据返回（因为会创建没有匹配行的WHERE子句）。这条SQL语句产生的输出如下：

输出▼

```
prod_name                 prod_price
-------------------       ----------
Fish bean bag toy         3.4900
Bird bean bag toy         3.4900
Rabbit bean bag toy       3.4900
8 inch teddy bear         5.9900
12 inch teddy bear        8.9900
18 inch teddy bear        11.9900
Raggedy Ann               4.9900
```

> **OR**
> WHERE子句中使用的关键字，用来表示检索匹配任一给定条件的行。

5.1.3 求值顺序

WHERE子句可以包含任意数目的AND和OR操作符。允许两者结合以进行复杂、高级的过滤。

但是，组合AND和OR会带来了一个有趣的问题。为了说明这个问题，来看一个例子。假如需要列出价格为10美元及以上，且由DLL01或BRS01制造的所有产品。下面的SELECT语句使用组合的AND和OR操作符建立了一个WHERE子句：

输入▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = ‘BRS01’
AND prod_price >= 10;
```

输出▼

```
prod_name                 prod_price
-------------------       ----------
Fish bean bag toy         3.4900
Bird bean bag toy         3.4900
Rabbit bean bag toy       3.4900
18 inch teddy bear        11.9900
```

```
Raggedy Ann                4.9900
```

分析▼

请看上面的结果。返回的行中有4行价格小于10美元，显然，返回的行未按预期的进行过滤。为什么会这样呢？原因在于求值的顺序。SQL（像多数语言一样）在处理OR操作符前，优先处理AND操作符。当SQL看到上述WHERE子句时，它理解为：由供应商BRS01制造的价格为10美元以上的所有产品，以及由供应商DLL01制造的所有产品，而不管其价格如何。换句话说，由于AND在求值过程中优先级更高，操作符被错误地组合了。

此问题的解决方法是使用圆括号对操作符进行明确分组。请看下面的SELECT语句及输出：

输入▼

```
SELECT prod_name, prod_price
FROM Products
WHERE (vend_id = 'DLL01' OR vend_id = ‘BRS01’)
AND prod_price >= 10;
```

输出▼

```
prod_name                  prod_price
-------------------        ----------
18 inch teddy bear         11.9900
```

分析▼

这条SELECT语句与前一条的唯一差别是，将前两个条件用圆括号括了起来。因为圆括号具有比AND或OR操作符更高的求值顺序，所以DBMS首先过滤圆括号内的OR条件。这时，SQL语句变成了选择由供应商DLL01或BRS01制造的且价格在10美元及以上的所有产品，这正是我们希望的结果。

> 提示：在**WHERE**子句中使用圆括号
> 任何时候使用具有AND和OR操作符的WHERE子句，都应该使用圆括号明确地分组操作符。不要过分依赖默认求值顺序，即使它确实如你希望的那样。使用圆括号没有什么坏处，它能消除歧义。

5.2 IN操作符

IN操作符用来指定条件范围，范围中的每个条件都可以进行匹配。IN取一组由逗号分隔、括在圆括号中的合法值。下面的例子说明了这个操作符：

输入▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id IN ( 'DLL01', 'BRS01' )
ORDER BY prod_name;
```

输出▼

```
prod_name                prod_price
-------------------      ----------
12 inch teddy bear       8.9900
18 inch teddy bear       11.9900
8 inch teddy bear        5.9900
Bird bean bag toy        3.4900
Fish bean bag toy        3.4900
Rabbit bean bag toy      3.4900
Raggedy Ann              4.9900
```

分析▼

此SELECT语句检索由供应商DLL01和BRS01制造的所有产品。IN操作符后跟由逗号分隔的合法值，这些值必须括在圆括号中。

你可能会猜测IN操作符完成了与OR相同的功能，恭喜你猜对了！下面的SQL语句完成与上面的例子相同的工作：

输入▼

```
SELECT prod_name, prod_price
FROM Products
WHERE vend_id = 'DLL01' OR vend_id = 'BRS01'
ORDER BY prod_name;
```

输出▼

```
prod_name                prod_price
-------------------      ----------
12 inch teddy bear       8.9900
```

```
18 inch teddy bear        11.9900
8 inch teddy bear         5.9900
Bird bean bag toy         3.4900
Fish bean bag toy         3.4900
Rabbit bean bag toy       3.4900
Raggedy Ann               4.9900
```

为什么要使用IN操作符？其优点为：

- 在有很多合法选项时，IN操作符的语法更清楚，更直观。
- 在与其他AND和OR操作符组合使用IN时，求值顺序更容易管理。
- IN操作符一般比一组OR操作符执行得更快（在上面这个合法选项很少的例子中，你看不出性能差异）。
- IN的最大优点是可以包含其他SELECT语句，能够更动态地建立WHERE子句。第11课会对此进行详细介绍。

> **IN**
> WHERE子句中用来指定要匹配值的清单的关键字，功能与OR相当。

5.3 NOT操作符

WHERE子句中的NOT操作符有且只有一个功能，那就是否定其后所跟的任何条件。因为NOT从不单独使用（它总是与其他操作符一起使用），所以它的语法与其他操作符有所不同。NOT关键字可以用在要过滤的列前，而不仅是在其后。

> **NOT**
> WHERE子句中用来否定其后条件的关键字。

下面的例子说明NOT的用法。为了列出除DLL01之外的所有供应商制造的产品，可编写如下的代码：

输入▼

```
SELECT prod_name
FROM Products
WHERE NOT vend_id = 'DLL01'
ORDER BY prod_name;
```

输出▼

```
prod_name
-------------------
12 inch teddy bear
18 inch teddy bear
8 inch teddy bear
King doll
Queen doll
```

分析▼
这里的NOT否定跟在其后的条件，因此，DBMS不是匹配vend_id为DLL01，而是匹配非DLL01之外的所有东西。

上面的例子也可以使用<>操作符来完成，如下所示：

输入▼

```
SELECT prod_name
FROM Products
WHERE vend_id <> 'DLL01'
ORDER BY prod_name;
```

输出▼

```
prod_name
-------------------
12 inch teddy bear
18 inch teddy bear
8 inch teddy bear
King doll
Queen doll
```

分析▼
为什么使用NOT？对于这里的这种简单的WHERE子句，使用NOT确实没有什么优势。但在更复杂的子句中，NOT是非常有用的。例如，在与IN操作符联合使用时，NOT可以非常简单地找出与条件列表不匹配的行。

> **说明：MariaDB中的NOT**
> MariaDB支持使用NOT否定IN、BETWEEN和EXISTS子句。大多数DBMS允许使用NOT否定任何条件。

5.4　小结

这一课讲授如何用AND和OR操作符组合成WHERE子句，还讲授了如何明确地管理求值顺序，如何使用IN和NOT操作符。

第6课　用通配符进行过滤

这一课介绍什么是通配符、如何使用通配符以及怎样使用`LIKE`操作符进行通配搜索，以便对数据进行复杂过滤。

6.1　LIKE操作符

前面介绍的所有操作符都是针对已知值进行过滤的。不管是匹配一个值还是多个值，检验大于还是小于已知值，或者检查某个范围的值，其共同点是过滤中使用的值都是已知的。

但是，这种过滤方法并不是任何时候都好用。例如，怎样搜索产品名中包含文本bean bag的所有产品？用简单的比较操作符肯定不行，必须使用通配符。利用通配符，可以创建比较特定数据的搜索模式。在这个例子中，如果你想找出名称包含bean bag的所有产品，可以构造一个通配符搜索模式，找出在产品名的任何位置出现bean bag的产品。

> 通配符（**wildcard**）
> 用来匹配值的一部分的特殊字符。

> 搜索模式（**search pattern**）
> 由字面值、通配符或两者组合构成的搜索条件。

通配符本身实际上是SQL的`WHERE`子句中有特殊含义的字符，SQL支持几种通配符。为在搜索子句中使用通配符，必须使用`LIKE`操作符。`LIKE`指示DBMS，后跟的搜索模式利用通配符匹配而不是简单的相等匹配进行比较。

> 谓词（**predicate**）
> 操作符何时不是操作符？答案是，它作为谓词时。从技术上说，`LIKE`是谓词而不是操作符。虽然最终的结果是相同的，但应该对此术语有所了解，以免在SQL文献或手册中遇到此术语时不知所云。

通配符搜索只能用于文本字段（字符串），非文本数据类型字段不能使用通配符搜索。

6.1.1　百分号（%）通配符

最常使用的通配符是百分号（%）。在搜索串中，%表示任何字符出现任意次数。例如，为了找出所有以词Fish起头的产品，可发布以下SELECT语句：

输入▼

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE 'Fish%';
```

输出▼

```
prod_id       prod_name
-------      ------------------
BNBG01         Fish bean bag toy
```

分析▼
此例子使用了搜索模式'Fish%'。在执行这条子句时，将检索任意以Fish起头的词。%告诉DBMS接受Fish之后的任意字符，不管它有多少字符。

> 说明：**Access通配符**
> 如果使用的是Microsoft Access，需要使用*而不是%。

> 说明：区分大小写
> 根据DBMS的不同及其配置，搜索可以是区分大小写的。如果区分大小写，则'fish%'与Fish bean bag toy就不匹配。

通配符可在搜索模式中的任意位置使用，并且可以使用多个通配符。下面的例子使用两个通配符，它们位于模式的两端：

输入▼

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '%bean bag%';
```

输出▼

```
prod_id        prod_name
---------      --------------------
BNBG01         Fish bean bag toy
BNBG02         Bird bean bag toy
BNBG03         Rabbit bean bag toy
```

分析▼

搜索模式'%bean bag%'表示匹配任何位置上包含文本bean bag的值，不论它之前或之后出现什么字符。

通配符也可以出现在搜索模式的中间，虽然这样做不太有用。下面的例子找出以F起头、以y结尾的所有产品：

输入▼

```
SELECT prod_name
FROM Products
WHERE prod_name LIKE 'F%y';
```

> 提示：根据部分信息搜索电子邮件地址
> 有一种情况下把通配符放在搜索模式中间是很有用的，就是根据邮件地址的一部分来查找电子邮件，例如WHERE email LIKE 'b%@forta.com'。

需要特别注意，除了能匹配一个或多个字符外，%还能匹配0个字符。%代表搜索模式中给定位置的0个、1个或多个字符。

> 说明：请注意后面所跟的空格
> 包括Access在内的许多DBMS都用空格来填补字段的内容。例如，如果某列有50个字符，而存储的文本为Fish bean bag toy（17个字符），则为填满该列需要在文本后附加33个空格。这样做一般对数据及其使用没有影响，但是可能对上述SQL语句有负面影响。子句WHERE prod_name LIKE 'F%y'只匹配以F开头、以y结尾的prod_name。如果值后面跟空格，则不是以y结尾，所以Fish bean bag toy就不会检索出来。简单的解决

> 办法是给搜索模式再增加一个%号：'F%y%'还匹配y之后的字符（或空格）。更好的解决办法是用函数去掉空格。请参阅第8课。

> 警告：请注意**NULL**
> 通配符%看起来像是可以匹配任何东西，但有个例外，这就是NULL。子句WHERE prod_name LIKE '%'不会匹配产品名称为NULL的行。

6.1.2　下划线（_）通配符

另一个有用的通配符是下划线（_）。下划线的用途与%一样，但它只匹配单个字符，而不是多个字符。

> 说明：**DB2**通配符
> DB2不支持通配符_。
>
> 说明：**Access**通配符
> 如果使用的是Microsoft Access，需要使用?而不是_。

举一个例子：

输入▼

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '__ inch teddy bear';
```

> 说明：请注意后面所跟的空格
> 与上例一样，可能需要给这个模式添加一个通配符。

输出▼

```
prod_id      prod_name
--------     --------------------
BR02         12 inch teddy bear
BR03         18 inch teddy bear
```

分析▼

这个WHERE子句中的搜索模式给出了后面跟有文本的两个通配符。结果只显示匹配搜索模式的行：第一行中下划线匹配12，第二行中匹配18。8 inch teddy bear产品没有匹配，因为搜索模式要求匹配两个通配符而不是一个。对照一下，下面的SELECT语句使用%通配符，返回三行产品：

输入▼

```
SELECT prod_id, prod_name
FROM Products
WHERE prod_name LIKE '% inch teddy bear';
```

输出▼

```
prod_id      prod_name
--------     --------------------
BR01         8 inch teddy bear
BR02         12 inch teddy bear
BNR3         18 inch teddy bear
```

与%能匹配多个字符不同，_总是刚好匹配一个字符，不能多也不能少。

6.1.3 方括号（[ ]）通配符

方括号（[]）通配符用来指定一个字符集，它必须匹配指定位置（通配符的位置）的一个字符。

> 说明：并不总是支持集合
> 与前面描述的通配符不一样，并不是所有DBMS都支持用来创建集合的[]。只有微软的Access和SQL Server支持集合。为确定你使用的DBMS是否支持集合，请参阅相应的文档。

例如，找出所有名字以J或M起头的联系人，可进行如下查询：

输入▼

```
SELECT cust_contact
FROM Customers
WHERE cust_contact LIKE '[JM]%'
```

```
ORDER BY cust_contact;
```

输出▼

```
cust_contact
-----------------
Jim Jones
John Smith
Michelle Green
```

分析▼
此语句的WHERE子句中的模式为'[JM]%'。这一搜索模式使用了两个不同的通配符。[JM]匹配方括号中任意一个字符，它也只能匹配单个字符。因此，任何多于一个字符的名字都不匹配。[JM]之后的%通配符匹配第一个字符之后的任意数目的字符，返回所需结果。

此通配符可以用前缀字符^（脱字号）来否定。例如，下面的查询匹配以J和 M 之外的任意字符起头的任意联系人名（与前一个例子相反）：

输入▼

```
SELECT cust_contact
FROM Customers
WHERE cust_contact LIKE '[^JM]%'
ORDER BY cust_contact;
```

> 说明：**Access**中的否定集合
> 如果使用的是Microsoft Access，需要用!而不是^来否定一个集合，因此，使用的是[!JM]而不是[^JM]。

当然，也可以使用NOT操作符得出类似的结果。^的唯一优点是在使用多个WHERE子句时可以简化语法：

输入▼

```
SELECT cust_contact
FROM Customers
WHERE NOT cust_contact LIKE '[JM]%'
ORDER BY cust_contact;
```

6.2 使用通配符的技巧

正如所见，SQL的通配符很有用。但这种功能是有代价的，即通配符搜索一般比前面讨论的其他搜索要耗费更长的处理时间。这里给出一些使用通配符时要记住的技巧。

- 不要过度使用通配符。如果其他操作符能达到相同的目的，应该使用其他操作符。
- 在确实需要使用通配符时，也尽量不要把它们用在搜索模式的开始处。把通配符置于开始处，搜索起来是最慢的。
- 仔细注意通配符的位置。如果放错地方，可能不会返回想要的数据。

总之，通配符是一种极其重要和有用的搜索工具，以后我们经常会用到它。

6.3 小结

这一课介绍了什么是通配符，如何在`WHERE`子句中使用SQL通配符，还说明了通配符应该细心使用，不要使用过度。

第7课　创建计算字段

这一课介绍什么是计算字段，如何创建计算字段，以及如何从应用程序中使用别名引用它们。

7.1　计算字段

存储在数据库表中的数据一般不是应用程序所需要的格式，下面举几个例子。

- 需要显示公司名，同时还需要显示公司的地址，但这两个信息存储在不同的表列中。
- 城市、州和邮政编码存储在不同的列中（应该这样），但邮件标签打印程序需要把它们作为一个有恰当格式的字段检索出来。
- 列数据是大小写混合的，但报表程序需要把所有数据按大写表示出来。
- 物品订单表存储物品的价格和数量，不存储每个物品的总价格（用价格乘以数量即可）。但为打印发票，需要物品的总价格。
- 需要根据表数据进行诸如总数、平均数的计算。

在上述每个例子中，存储在表中的数据都不是应用程序所需要的。我们需要直接从数据库中检索出转换、计算或格式化过的数据，而不是检索出数据，然后再在客户端应用程序中重新格式化。

这就是计算字段可以派上用场的地方了。与前几课介绍的列不同，计算字段并不实际存在于数据库表中。计算字段是运行时在`SELECT`语句内创建的。

> 字段（**field**）
> 基本上与列（column）的意思相同，经常互换使用，不过数据库列一般称为列，而术语字段通常与计算字段一起使用。

需要特别注意，只有数据库知道`SELECT`语句中哪些列是实际的表列，哪些列是计算字段。从客户端（如应用程序）来看，计算字段的数据与其他列的数据的返回方式相同。

> 提示：客户端与服务器的格式
> 在SQL语句内可完成的许多转换和格式化工作都可以直接在客户端应用程序内完成。但一般来说，在数据库服务器上完成这些操作比在客户端中完成要快得多。

7.2　拼接字段

为了说明如何使用计算字段，我们来举一个简单例子，创建由两列组成的标题。

`Vendors`表包含供应商名和地址信息。假如要生成一个供应商报表，需要在格式化的名称（位置）中列出供应商的位置。

此报表需要一个值，而表中数据存储在两个列`vend_name`和`vend_country`中。此外，需要用括号将`vend_country`括起来，这些东西都没有存储在数据库表中。这个返回供应商名称和地址的`SELECT`语句很简单，但我们是如何创建这个组合值的呢？

> 拼接（**concatenate**）
> 将值联结到一起（将一个值附加到另一个值）构成单个值。

解决办法是把两个列拼接起来。在SQL中的`SELECT`语句中，可使用一个特殊的操作符来拼接两个列。根据你所使用的DBMS，此操作符可用加号（+）或两个竖杠（||）表示。在MySQL和MariaDB中，必须使用特殊的函数。

> 说明：是**+**还是**||**？
> Access和SQL Server使用+号。DB2、Oracle、PostgreSQL、SQLite和Open Office Base使用||。详细请参阅具体的DBMS文档。

下面是使用加号的例子（多数DBMS使用这种语法）：

输入▼

```
SELECT vend_name + ' (' + vend_country + ')'
FROM Vendors
ORDER BY vend_name;
```

输出▼

```
-----------------------------------------------------------
Bear Emporium                                (USA        )
Bears R Us                                   (USA        )
Doll House Inc.                              (USA        )
Fun and Games                                (England    )
Furball Inc.                                 (USA        )
Jouets et ours                               (France     )
```

下面是相同的语句，但使用的是||语法：

输入▼

```
SELECT vend_name || ' (' || vend_country || ')'
FROM Vendors
ORDER BY vend_name;
```

输出▼

```
-----------------------------------------------------------
Bear Emporium                                (USA        )
Bears R Us                                   (USA        )
Doll House Inc.                              (USA        )
Fun and Games                                (England    )
Furball Inc.                                 (USA        )
Jouets et ours                               (France     )
```

分析▼
上面两个SELECT语句拼接以下元素：

- 存储在vend_name列中的名字；
- 包含一个空格和一个左圆括号的字符串；
- 存储在vend_country列中的国家；
- 包含一个右圆括号的字符串。

从上述输出中可以看到，SELECT语句返回包含上述四个元素的一个列（计算字段）。

再看看上述SELECT语句返回的输出。结合成一个计算字段的两个列用空格填充。许多数据库（不是所有）保存填充为列宽的文本值，而实际上你要的结果不需要这些空格。为正确返回格式化的数据，必须

去掉这些空格。这可以使用SQL的RTRIM()函数来完成，如下所示：

输入▼

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
FROM Vendors
ORDER BY vend_name;
```

输出▼

```
-----------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

下面是相同的语句，但使用的是||：

输入▼

```
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
FROM Vendors
ORDER BY vend_name;
```

输出▼

```
-----------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

分析▼

RTRIM()函数去掉值右边的所有空格。通过使用RTRIM()，各个列都进行了整理。

> 说明：**TRIM函数**
> 大多数DBMS都支持RTRIM()（正如刚才所见，它去掉字符串右

边的空格）、LTRIM()（去掉字符串左边的空格）以及TRIM()（去掉字符串左右两边的空格）。

使用别名

从前面的输出可以看到，SELECT语句可以很好地拼接地址字段。但是，这个新计算列的名字是什么呢？实际上它没有名字，它只是一个值。如果仅在SQL查询工具中查看一下结果，这样没有什么不好。但是，一个未命名的列不能用于客户端应用中，因为客户端没有办法引用它。

为了解决这个问题，SQL支持列别名。别名（alias）是一个字段或值的替换名。别名用AS关键字赋予。请看下面的SELECT语句：

输入▼

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

输出▼

```
vend_title
-----------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

下面是相同的语句，但使用的是||语法：

输入▼

```
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

下面是MySQL和MariaDB中使用的语句：

输入▼

```
SELECT Concat(vend_name, ' (', vend_country, ')')
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

分析▼

SELECT语句本身与以前使用的相同，只不过这里的计算字段之后跟了文本AS vend_title。它指示SQL创建一个包含指定计算结果的名为vend_title的计算字段。从输出可以看到，结果与以前的相同，但现在列名为vend_title，任何客户端应用都可以按名称引用这个列，就像它是一个实际的表列一样。

> 说明：**AS**通常可选
> 在很多DBMS中，AS关键字是可选的，不过最好使用它，这被视为一条最佳实践。
>
> 提示：别名的其他用途
> 别名还有其他用途。常见的用途包括在实际的表列名包含不合法的字符（如空格）时重新命名它，在原来的名字含混或容易误解时扩充它。

> 注意：别名
> 别名的名字既可以是一个单词，也可以是一个字符串。如果是后者，字符串应该括在引号中。虽然这种做法是合法的，但不建议这么去做。多单词的名字可读性高，不过会给客户端应用带来各种问题。因此，别名最常见的使用是将多个单词的列名重命名为一个单词的名字。

> 说明：导出列
> 别名有时也称为导出列（derived column），不管怎么叫，它们所代表的是相同的东西。

7.3 执行算术计算

计算字段的另一常见用途是对检索出的数据进行算术计算。举个例子，Orders表包含收到的所有订单，OrderItems表包含每个订单中的各项物品。下面的SQL语句检索订单号20008中的所有物品：

输入▼

```
SELECT prod_id, quantity, item_price
FROM OrderItems
WHERE order_num = 20008;
```

输出▼

```
prod_id         quantity        item_price
----------      -----------     ---------------------
RGAN01          5               4.9900
BR03            5               11.9900
BNBG01          10              3.4900
BNBG02          10              3.4900
BNBG03          10              3.4900
```

item_price列包含订单中每项物品的单价。如下汇总物品的价格（单价乘以订购数量）：

输入▼

```
SELECT prod_id,
       quantity,
       item_price,
       quantity*item_price AS expanded_price
FROM OrderItems
WHERE order_num = 20008;
```

输出▼

```
prod_id         quantity        item_price      expanded_pric
----------      -----------     ------------    --------------
RGAN01          5               4.9900          24.9500
BR03            5               11.9900         59.9500
BNBG01          10              3.4900          34.9000
```

BNBG02	10	3.4900	34.9000
BNBG03	10	3.4900	34.9000

分析▼

输出中显示的expanded_price列是一个计算字段，此计算为quantity*item_price。客户端应用现在可以使用这个新计算列，就像使用其他列一样。

SQL支持表7-1中列出的基本算术操作符。此外，圆括号可用来区分优先顺序。关于优先顺序的介绍，请参阅第5课。

表7-1 SQL算术操作符

操作符	说明
+	加
-	减
*	乘
/	除

> 提示：如何测试计算
> SELECT语句为测试、检验函数和计算提供了很好的方法。虽然SELECT通常用于从表中检索数据，但是省略了FROM子句后就是简单地访问和处理表达式，例如SELECT 3 * 2;将返回6，SELECT Trim(' abc ');将返回abc，SELECT Now();使用Now()函数返回当前日期和时间。现在你明白了，可以根据需要使用SELECT语句进行检验。

7.4 小结

这一课介绍了计算字段以及如何创建计算字段。我们用例子说明了计算字段在字符串拼接和算术计算中的用途。此外，还讲述了如何创建和使用别名，以便应用程序能引用计算字段。

第8课　使用函数处理数据

这一课介绍什么是函数，DBMS支持何种函数，以及如何使用这些函数；还将讲解为什么SQL函数的使用可能会带来问题。

8.1　函数

与大多数其他计算机语言一样，SQL也可以用函数来处理数据。函数一般是在数据上执行的，为数据的转换和处理提供了方便。

前一课中用来去掉字符串尾的空格的`RTRIM()`就是一个函数。

函数带来的问题

在学习这一课并进行实践之前，你应该了解使用SQL函数所存在的问题。

与几乎所有DBMS都等同地支持SQL语句（如`SELECT`）不同，每一个DBMS都有特定的函数。事实上，只有少数几个函数被所有主要的DBMS等同地支持。虽然所有类型的函数一般都可以在每个DBMS中使用，但各个函数的名称和语法可能极其不同。为了说明可能存在的问题，表8-1列出了3个常用的函数及其在各个DBMS中的语法：

表8-1 DBMS函数的差异

函　　数	语　　法
提取字符串的组成部分	Access使用MID()；DB2、Oracle、PostgreSQL和SQLite使用SUBSTR()；MySQL和SQL Server使用SUBSTRING()
数据类型转换	Access和Oracle使用多个函数，每种类型的转换有一个函数；DB2和PostgreSQL使用CAST()；MariaDB、MySQL和SQL Server使用CONVERT()
取当前日期	Access使用NOW()；DB2和PostgreSQL使用CURRENT_DATE；MariaDB和MySQL使用CURDATE()；Oracle使用SYSDATE；SQL Server使用GETDATE()；SQLite使用DATE()

可以看到，与SQL语句不一样，SQL函数不是可移植的。这意味着为

特定SQL实现编写的代码在其他实现中可能不正常。

> 可移植（**portable**）
> 所编写的代码可以在多个系统上运行。

为了代码的可移植，许多SQL程序员不赞成使用特定于实现的功能。虽然这样做很有好处，但有的时候并不利于应用程序的性能。如果不使用这些函数，编写某些应用程序代码会很艰难。必须利用其他方法来实现DBMS可以非常有效完成的工作。

> 提示：是否应该使用函数？
> 现在，你面临是否应该使用函数的选择。决定权在你，使用或是不使用也没有对错之分。如果你决定使用函数，应该保证做好代码注释，以便以后你（或其他人）能确切地知道所编写的SQL代码的含义。

8.2　使用函数

大多数SQL实现支持以下类型的函数。

- 用于处理文本字符串（如删除或填充值，转换值为大写或小写）的文本函数。
- 用于在数值数据上进行算术操作（如返回绝对值，进行代数运算）的数值函数。
- 用于处理日期和时间值并从这些值中提取特定成分（如返回两个日期之差，检查日期有效性）的日期和时间函数。
- 返回DBMS正使用的特殊信息（如返回用户登录信息）的系统函数。

我们在上一课看到函数用于`SELECT`后面的列名，但函数的作用不仅于此。它还可以作为`SELECT`语句的其他成分，如在`WHERE`子句中使用，在其他SQL语句中使用等，后面会做更多的介绍。

8.2.1　文本处理函数

在上一课，我们已经看过一个文本处理函数的例子，其中使用`RTRIM()`函数来去除列值右边的空格。下面是另一个例子，这次使用的是`UPPER()`函数：

输入▼

```
SELECT vend_name, UPPER(vend_name) AS vend_name_upcase
FROM Vendors
ORDER BY vend_name;
```

输出▼

```
vend_name                        vend_name_upcase
---------------------------      ----------------------------
Bear Emporium                    BEAR EMPORIUM
Bears R Us                       BEARS R US
Doll House Inc.                  DOLL HOUSE INC.
Fun and Games                    FUN AND GAMES
Furball Inc.                     FURBALL INC.
Jouets et ours                   JOUETS ET OURS
```

分析▼

可以看到，UPPER()将文本转换为大写，因此本例子中每个供应商都列出两次，第一次为Vendors表中存储的值，第二次作为列vend_name_upcase转换为大写。

表8-2列出了一些常用的文本处理函数。

表**8-2** 常用的文本处理函数

函　　数	说　　明
LEFT()（或使用子字符串函数）	返回字符串左边的字符
LENGTH()（也使用DATALENGTH()或LEN()）	返回字符串的长度
LOWER()（Access使用LCASE()）	将字符串转换为小写
LTRIM()	去掉字符串左边的空格
RIGHT()（或使用子字符串函数）	返回字符串右边的字符
RTRIM()	去掉字符串右边的空格
SOUNDEX()	返回字符串的SOUNDEX值
UPPER()（Access使用UCASE()）	将字符串转换为大写

表8-2中的SOUNDEX需要做进一步的解释。SOUNDEX是一个将任何文本串转换为描述其语音表示的字母数字模式的算法。SOUNDEX考虑了类似的发音字符和音节，使得能对字符串进行发音比较而不是字母比较。虽然SOUNDEX不是SQL概念，但多数DBMS都提供对SOUNDEX的支持。

> 说明：**SOUNDEX**支持
> Microsoft Access和PostgreSQL不支持`SOUNDEX()`，因此以下的例子不适用于这些DBMS。
>
> 另外，如果在创建SQLite时使用了`SQLITE_SOUNDEX`编译时选项，那么`SOUNDEX()`在SQLite中就可用。因为`SQLITE_SOUNDEX`不是默认的编译时选项，所以多数SQLite实现不支持`SOUNDEX()`。

下面给出一个使用`SOUNDEX()`函数的例子。`Customers`表中有一个顾客`Kids Place`，其联系名为`Michelle Green`。但如果这是错误的输入，此联系名实际上应该是`Michael Green`，该怎么办呢？显然，按正确的联系名搜索不会返回数据，如下所示：

输入▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_contact = 'Michael Green';
```

输出▼

```
cust_name                     cust_contact
--------------------------    ------------------------------
```

现在试一下使用`SOUNDEX()`函数进行搜索，它匹配所有发音类似于`Michael Green`的联系名：

输入▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE SOUNDEX(cust_contact) = SOUNDEX('Michael Green');
```

输出▼

```
cust_name                     cust_contact
--------------------------    ------------------------------
Kids Place                    Michelle Green
```

分析▼

在这个例子中，WHERE子句使用SOUNDEX()函数把cust_contact列值和搜索字符串转换为它们的SOUNDEX值。因为Michael Green和Michelle Green发音相似，所以它们的SOUNDEX值匹配，因此WHERE子句正确地过滤出了所需的数据。

8.2.2　日期和时间处理函数

日期和时间采用相应的数据类型存储在表中，每种DBMS都有自己的特殊形式。日期和时间值以特殊的格式存储，以便能快速和有效地排序或过滤，并且节省物理存储空间。

应用程序一般不使用日期和时间的存储格式，因此日期和时间函数总是用来读取、统计和处理这些值。由于这个原因，日期和时间函数在SQL中具有重要的作用。遗憾的是，它们很不一致，可移植性最差。

我们举个简单的例子，来说明日期处理函数的用法。Orders表中包含的订单都带有订单日期。为在SQL Server中检索2012年的所有订单，可如下进行：

输入▼

```
SELECT order_num
FROM Orders
WHERE DATEPART(yy, order_date) = 2012;
```

输出▼

```
order_num
-----------
20005
20006
20007
20008
20009
```

在Access中使用如下版本：

输入▼

```
SELECT order_num
```

```
FROM Orders
WHERE DATEPART('yyyy', order_date) = 2012;
```

分析▼

这个例子（SQL Server和Sybase版本以及Access版本）使用了DATEPART()函数，顾名思义，此函数返回日期的某一部分。DATEPART()函数有两个参数，它们分别是返回的成分和从中返回成分的日期。在此例子中，DATEPART()只从order_date列中返回年份。通过与2012比较，WHERE子句只过滤出此年份的订单。

下面是使用名为DATE_PART()的类似函数的PostgreSQL版本：

输入▼

```
SELECT order_num
FROM Orders
WHERE DATE_PART('year', order_date) = 2012;
```

Oracle没有DATEPART()函数，不过有几个可用来完成相同检索的日期处理函数。例如：

输入▼

```
SELECT order_num
FROM Orders
WHERE to_number(to_char(order_date, 'YYYY')) = 2012;
```

分析▼

在这个例子中，to_char()函数用来提取日期的成分，to_number()用来将提取出的成分转换为数值，以便能与2012进行比较。

完成相同工作的另一方法是使用BETWEEN操作符：

输入▼

```
SELECT order_num
FROM Orders
WHERE order_date BETWEEN to_date('01-01-2012')
AND to_date('12-31-2012');
```

分析▼

在此例子中，Oracle的`to_date()`函数用来将两个字符串转换为日期。一个包含2012年1月1日，另一个包含2012年12月31日。`BETWEEN`操作符用来找出两个日期之间的所有订单。值得注意的是，相同的代码在SQL Server中不起作用，因为它不支持`to_date()`函数。但是，如果用`CONVERT()`替换`to_date()`，当然可以使用这种类型的语句。

MySQL和MariaDB具有各种日期处理函数，但没有`DATEPART()`。MySQL和MariaDB用户可使用名为`YEAR()`的函数从日期中提取年份：

输入▼

```
SELECT order_num
FROM Orders
WHERE YEAR(order_date) = 2012;
```

在SQLite中有个小技巧：

输入▼

```
SELECT order_num
FROM Orders
WHERE strftime('%Y', order_date) = '2012';
```

这里给出的例子提取和使用日期的成分（年）。按月份过滤，可以进行相同的处理，使用`AND`操作符可以进行年和月份的比较。

DBMS提供的功能远不只简单的日期成分提取。大多数DBMS具有比较日期、执行基于日期的运算、选择日期格式等的函数。但是，可以看到，不同DBMS的日期-时间处理函数可能不同。关于具体DBMS支持的日期-时间处理函数，请参阅相应的文档。

8.2.3　数值处理函数

数值处理函数仅处理数值数据。这些函数一般主要用于代数、三角或几何运算，因此不像字符串或日期-时间处理函数使用那么频繁。

具有讽刺意味的是，在主要DBMS的函数中，数值函数是最一致、最统一的函数。表8-3列出一些常用的数值处理函数。

表8-3 常用数值处理函数

函　　数	说　　明
ABS()	返回一个数的绝对值
COS()	返回一个角度的余弦
EXP()	返回一个数的指数值
PI()	返回圆周率
SIN()	返回一个角度的正弦
SQRT()	返回一个数的平方根
TAN()	返回一个角度的正切

关于具体DBMS所支持的算术处理函数，请参阅相应的文档。

8.3　小结

这一课介绍了如何使用SQL的数据处理函数。虽然这些函数在格式化、处理和过滤数据中非常有用，但它们在各种SQL实现中很不一致（SQL Server和Oracle之间的差异说明了这一点）。

第9课　汇总数据

这一课介绍什么是SQL的聚集函数，如何利用它们汇总表的数据。

9.1　聚集函数

我们经常需要汇总数据而不用把它们实际检索出来，为此SQL提供了专门的函数。使用这些函数，SQL查询可用于检索数据，以便分析和报表生成。这种类型的检索例子有：

- 确定表中行数（或者满足某个条件或包含某个特定值的行数）；
- 获得表中某些行的和；
- 找出表列（或所有行或某些特定的行）的最大值、最小值、平均值。

上述例子都需要汇总表中的数据，而不需要实际数据本身。因此，返回实际表数据纯属浪费时间和处理资源（更不用说带宽了）。再说一遍，我们实际想要的是汇总信息。

为方便这种类型的检索，SQL给出了5个聚集函数，见表9-1。这些函数能进行上述检索。与前一章介绍的数据处理函数不同，SQL的聚集函数在各种主要SQL实现中得到了相当一致的支持。

> 聚集函数（**aggregate function**）对某些行运行的函数，计算并返回一个值。

表9-1 SQL聚集函数

函　　数	说　　明
AVG()	返回某列的平均值
COUNT()	返回某列的行数
MAX()	返回某列的最大值
MIN()	返回某列的最小值
SUM()	返回某列值之和

以下说明各函数的使用。

9.1.1 AVG()函数

AVG()通过对表中行数计数并计算其列值之和，求得该列的平均值。AVG()可用来返回所有列的平均值，也可以用来返回特定列或行的平均值。

下面的例子使用AVG()返回Products表中所有产品的平均价格：

输入▼

```
SELECT AVG(prod_price) AS avg_price
FROM Products;
```

输出▼

```
avg_price
-------------
6.823333
```

分析▼

此SELECT语句返回值avg_price，它包含Products表中所有产品的平均价格。如第7课所述，avg_price是一个别名。

AVG()也可以用来确定特定列或行的平均值。下面的例子返回特定供应商所提供产品的平均价格：

输入▼

```
SELECT AVG(prod_price) AS avg_price
FROM Products
WHERE vend_id = 'DLL01';
```

输出▼

```
avg_price
-----------
3.8650
```

分析▼

这条SELECT语句与前一条的不同之处在于，它包含了WHERE子句。此WHERE子句仅过滤出vend_id为DLL01的产品，因此avg_price

中返回的值只是该供应商产品的平均值。

> 警告：只用于单个列
> **AVG()**只能用来确定特定数值列的平均值，而且列名必须作为函数参数给出。为了获得多个列的平均值，必须使用多个AVG()函数。

> 说明：**NULL**值
> AVG()函数忽略列值为NULL的行。

9.1.2　COUNT()函数

COUNT()函数进行计数。可利用COUNT()确定表中行的数目或符合特定条件的行的数目。

COUNT()函数有两种使用方式：

- 使用COUNT(*)对表中行的数目进行计数，不管表列中包含的是空值（NULL）还是非空值。
- 使用COUNT(column)对特定列中具有值的行进行计数，忽略NULL值。

下面的例子返回Customers表中顾客的总数：

输入▼

```
SELECT COUNT(*) AS num_cust
FROM Customers;
```

输出▼

```
num_cust
--------
5
```

分析▼
在此例子中，利用COUNT(*)对所有行计数，不管行中各列有什么值。计数值在num_cust中返回。

下面的例子只对具有电子邮件地址的客户计数：

输入▼

```
SELECT COUNT(cust_email) AS num_cust
FROM Customers;
```

输出▼

```
num_cust
--------
3
```

分析▼
这条SELECT语句使用COUNT(cust_email)对cust_email列中有值的行进行计数。在此例子中，cust_email的计数为3（表示5个顾客中只有3个顾客有电子邮件地址）。

> 说明：**NULL**值
> 如果指定列名，则COUNT()函数会忽略指定列的值为空的行，但如果COUNT()函数中用的是星号（*），则不忽略。

9.1.3 MAX()函数

MAX()返回指定列中的最大值。MAX()要求指定列名，如下所示：

输入▼

```
SELECT MAX(prod_price) AS max_price
FROM Products;
```

输出▼

```
max_price
----------
11.9900
```

分析▼
这里，MAX()返回Products表中最贵物品的价格。

> 提示：对非数值数据使用**MAX()**
> 虽然MAX()一般用来找出最大的数值或日期值，但许多（并非所有）DBMS允许将它用来返回任意列中的最大值，包括返回文本列中的最大值。在用于文本数据时，MAX()返回按该列排序后的最后一行。

> 说明：**NULL**值
> MAX()函数忽略列值为NULL的行。

9.1.4 MIN()函数

MIN()的功能正好与MAX()功能相反，它返回指定列的最小值。与MAX()一样，MIN()要求指定列名，如下所示：

输入▼

```
SELECT MIN(prod_price) AS min_price
FROM Products;
```

输出▼

```
min_price
----------
3.4900
```

分析▼
其中MIN()返回Products表中最便宜物品的价格。

> 提示：对非数值数据使用**MIN()**
> 虽然MIN()一般用来找出最小的数值或日期值，但许多（并非所有）DBMS允许将它用来返回任意列中的最小值，包括返回文本列中的最小值。在用于文本数据时，MIN()返回该列排序后最前面的行。

> 说明：**NULL**值

MIN()函数忽略列值为NULL的行。

9.1.5 SUM()函数

SUM()用来返回指定列值的和（总计）。下面举一个例子，OrderItems包含订单中实际的物品，每个物品有相应的数量。可如下检索所订购物品的总数（所有quantity值之和）：

输入▼

```
SELECT SUM(quantity) AS items_ordered
FROM OrderItems
WHERE order_num = 20005;
```

输出▼

```
items_ordered
----------
200
```

分析▼
函数SUM(quantity)返回订单中所有物品数量之和，WHERE子句保证只统计某个物品订单中的物品。

SUM()也可以用来合计计算值。在下面的例子中，合计每项物品的item_price*quantity，得出总的订单金额：

输入▼

```
SELECT SUM(item_price*quantity) AS total_price
FROM OrderItems
WHERE order_num = 20005;
```

输出▼

```
total_price
----------
1648.0000
```

分析▼

函数SUM(item_price*quantity)返回订单中所有物品价钱之和，WHERE子句同样保证只统计某个物品订单中的物品。

> 提示：在多个列上进行计算
> 如本例所示，利用标准的算术操作符，所有聚集函数都可用来执行多个列上的计算。

> 说明：**NULL**值
> SUM()函数忽略列值为NULL的行。

9.2　聚集不同值

以上5个聚集函数都可以如下使用：

- 对所有行执行计算，指定ALL参数或不指定参数（因为ALL是默认行为）。
- 只包含不同的值，指定DISTINCT参数。

> 提示：**ALL**为默认
> ALL参数不需要指定，因为它是默认行为。如果不指定DISTINCT，则假定为ALL。

> 说明：不要在**Access**中使用
> Microsoft Access在聚集函数中不支持DISTINCT，因此下面的例子不适合于Access。要在Access得到类似的结果，需要使用子查询把DISTINCT数据返回到外部SELECT COUNT(*)语句。

下面的例子使用AVG()函数返回特定供应商提供的产品的平均价格。它与上面的SELECT语句相同，但使用了DISTINCT参数，因此平均值只考虑各个不同的价格：

输入▼

```
SELECT AVG(DISTINCT prod_price) AS avg_price
```

```
FROM Products
WHERE vend_id = 'DLL01';
```

输出▼

```
avg_price
-----------
4.2400
```

分析▼
可以看到，在使用了DISTINCT后，此例子中的avg_price比较高，因为有多个物品具有相同的较低价格。排除它们提升了平均价格。

> 警告：**DISTINCT**不能用于**COUNT(*)**
> 如果指定列名，则DISTINCT只能用于COUNT()。DISTINCT不能用于COUNT(*)。类似地，DISTINCT必须使用列名，不能用于计算或表达式。

> 提示：将**DISTINCT**用于**MIN()**和**MAX()**
> 虽然DISTINCT从技术上可用于MIN()和MAX()，但这样做实际上没有价值。一个列中的最小值和最大值不管是否只考虑不同值，结果都是相同的。

> 说明：其他聚集参数
> 除了这里介绍的DISTINCT和ALL参数，有的DBMS还支持其他参数，如支持对查询结果的子集进行计算的TOP和TOP PERCENT。为了解具体的DBMS支持哪些参数，请参阅相应的文档。

9.3　组合聚集函数

目前为止的所有聚集函数例子都只涉及单个函数。但实际上，SELECT语句可根据需要包含多个聚集函数。请看下面的例子：

输入▼

```
SELECT COUNT(*) AS num_items,
```

```
       MIN(prod_price) AS price_min,
       MAX(prod_price) AS price_max,
       AVG(prod_price) AS price_avg
FROM Products;
```

输出▼

```
num_items      price_min          price_max p        rice_av
----------     ---------------    ---------------    -------
9              3.4900             11.9900            6.8233
```

分析▼

这里用单条SELECT语句执行了4个聚集计算，返回4个值（Products表中物品的数目，产品价格的最高值、最低值以及平均值）。

> 警告：取别名
>
> 在指定别名以包含某个聚集函数的结果时，不应该使用表中实际的列名。虽然这样做也算合法，但许多SQL实现不支持，可能会产生模糊的错误消息。

9.4 小结

聚集函数用来汇总数据。SQL支持5个聚集函数，可以用多种方法使用它们，返回所需的结果。这些函数很高效，它们返回结果一般比你在自己的客户端应用程序中计算要快得多。

第10课　分组数据

这一课介绍如何分组数据，以便汇总表内容的子集。这涉及两个新SELECT语句子句：GROUP BY子句和HAVING子句。

10.1　数据分组

从上一课得知，使用SQL聚集函数可以汇总数据。这样，我们就能够对行进行计数，计算和与平均数，不检索所有数据就获得最大值和最小值。

目前为止的所有计算都是在表的所有数据或匹配特定的WHERE子句的数据上进行的。比如下面的例子返回供应商DLL01提供的产品数目：

输入▼

```
SELECT COUNT(*) AS num_prods
FROM Products
WHERE vend_id = 'DLL01';
```

输出▼

```
num_prods
-----------
4
```

如果要返回每个供应商提供的产品数目，该怎么办？或者返回只提供一项产品的供应商的产品，或者返回提供10个以上产品的供应商的产品，怎么办？

这就是分组大显身手的时候了。使用分组可以将数据分为多个逻辑组，对每个组进行聚集计算。

10.2　创建分组

分组是使用SELECT语句的GROUP BY子句建立的。理解分组的最好办法是看一个例子：

输入▼

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
GROUP BY vend_id;
```

输出▼

```
vend_id     num_prods
-------     ---------
BRS01       3
DLL01       4
FNG01       2
```

分析▼

上面的SELECT语句指定了两个列：vend_id包含产品供应商的ID，num_prods为计算字段（用COUNT(*)函数建立）。GROUP BY子句指示DBMS按vend_id排序并分组数据。这就会对每个vend_id而不是整个表计算num_prods一次。从输出中可以看到，供应商BRS01有3个产品，供应商DLL01有4个产品，而供应商FNG01有2个产品。

因为使用了GROUP BY，就不必指定要计算和估值的每个组了。系统会自动完成。GROUP BY子句指示DBMS分组数据，然后对每个组而不是整个结果集进行聚集。

在使用GROUP BY子句前，需要知道一些重要的规定。

- GROUP BY子句可以包含任意数目的列，因而可以对分组进行嵌套，更细致地进行数据分组。
- 如果在GROUP BY子句中嵌套了分组，数据将在最后指定的分组上进行汇总。换句话说，在建立分组时，指定的所有列都一起计算（所以不能从个别的列取回数据）。
- GROUP BY子句中列出的每一列都必须是检索列或有效的表达式（但不能是聚集函数）。如果在SELECT中使用表达式，则必须在GROUP BY子句中指定相同的表达式。不能使用别名。
- 大多数SQL实现不允许GROUP BY列带有长度可变的数据类型（如文本或备注型字段）。
- 除聚集计算语句外，SELECT语句中的每一列都必须在GROUP BY子句中给出。
- 如果分组列中包含具有NULL值的行，则NULL将作为一个分组返回。如果列中有多行NULL值，它们将分为一组。

- GROUP BY子句必须出现在WHERE子句之后，ORDER BY子句之前。

> 提示：**ALL**子句
> Microsoft SQL Server等有些SQL实现在GROUP BY中支持可选的ALL子句。这个子句可用来返回所有分组，即使是没有匹配行的分组也返回（在此情况下，聚集将返回NULL）。具体的DBMS是否支持ALL，请参阅相应的文档。

> 警告：通过相对位置指定列
> 有的SQL实现允许根据SELECT列表中的位置指定GROUP BY的列。例如，GROUP BY 2, 1可表示按选择的第二个列分组，然后再按第一个列分组。虽然这种速记语法很方便，但并非所有SQL实现都支持，并且使用它容易在编辑SQL语句时出错。

10.3　过滤分组

除了能用GROUP BY分组数据外，SQL还允许过滤分组，规定包括哪些分组，排除哪些分组。例如，你可能想要列出至少有两个订单的所有顾客。为此，必须基于完整的分组而不是个别的行进行过滤。

我们已经看到了WHERE子句的作用（第4课提及）。但是，在这个例子中WHERE不能完成任务，因为WHERE过滤指定的是行而不是分组。事实上，WHERE没有分组的概念。

那么，不使用WHERE使用什么呢？SQL为此提供了另一个子句，就是HAVING子句。HAVING非常类似于WHERE。事实上，目前为止所学过的所有类型的WHERE子句都可以用HAVING来替代。唯一的差别是，WHERE过滤行，而HAVING过滤分组。

> 提示：**HAVING**支持所有**WHERE**操作符
> 在第4课和第5课中，我们学习了WHERE子句的条件（包括通配符条件和带多个操作符的子句）。学过的这些有关WHERE的所有技术和选项都适用于HAVING。它们的句法是相同的，只是关键字有差别。

那么，怎么过滤分组呢？请看以下的例子：

输入▼

```
SELECT cust_id, COUNT(*) AS orders
FROM Orders
GROUP BY cust_id
HAVING COUNT(*) >= 2;
```

输出▼

```
cust_id         orders
----------      -----------
1000000001      2
```

分析▼

这条SELECT语句的前三行类似于上面的语句。最后一行增加了HAVING子句，它过滤COUNT(*) >= 2（两个以上订单）的那些分组。

可以看到，WHERE子句在这里不起作用，因为过滤是基于分组聚集值，而不是特定行的值。

> 说明：**HAVING**和**WHERE**的差别
> 这里有另一种理解方法，WHERE在数据分组前进行过滤，HAVING在数据分组后进行过滤。这是一个重要的区别，WHERE排除的行不包括在分组中。这可能会改变计算值，从而影响HAVING子句中基于这些值过滤掉的分组。

那么，有没有在一条语句中同时使用WHERE和HAVING子句的需要呢？事实上，确实有。假如想进一步过滤上面的语句，使它返回过去12个月内具有两个以上订单的顾客。为此，可增加一条WHERE子句，过滤出过去12个月内下过的订单，然后再增加HAVING子句过滤出具有两个以上订单的分组。

为了更好地理解，来看下面的例子，它列出具有两个以上产品且其价格大于等于4的供应商：

输入▼

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
WHERE prod_price >= 4
```

```
GROUP BY vend_id
HAVING COUNT(*) >= 2;
```

输出▼

```
vend_id     num_prods
-------     -----------
BRS01       3
FNG01       2
```

分析▼
这条语句中，第一行是使用了聚集函数的基本SELECT语句，很像前面的例子。WHERE子句过滤所有prod_price至少为4的行，然后按vend_id分组数据，HAVING子句过滤计数为2或2以上的分组。如果没有WHERE子句，就会多检索出一行（供应商DLL01，销售4个产品，价格都在4以下）：

输入▼

```
SELECT vend_id, COUNT(*) AS num_prods
FROM Products
GROUP BY vend_id
HAVING COUNT(*) >= 2;
```

输出▼

```
vend_id     num_prods
-------     -----------
BRS01       3
DLL01       4
FNG01       2
```

> **说明：使用HAVING和WHERE**
> HAVING与WHERE非常类似，如果不指定GROUP BY，则大多数DBMS会同等对待它们。不过，你自己要能区分这一点。使用HAVING时应该结合GROUP BY子句，而WHERE子句用于标准的行级过滤。

10.4　分组和排序

GROUP BY和ORDER BY经常完成相同的工作，但它们非常不同，理解这一点很重要。表10-1汇总了它们之间的差别。

表10-1 ORDER BY与GROUP BY

ORDER BY	GROUP BY
对产生的输出排序	对行分组，但输出可能不是分组的顺序
任意列都可以使用（甚至非选择的列也可以使用）	只可能使用选择列或表达式列，而且必须使用每个选择列表达式
不一定需要	如果与聚集函数一起使用列（或表达式），则必须使用

表10-1中列出的第一项差别极为重要。我们经常发现，用GROUP BY分组的数据确实是以分组顺序输出的。但并不总是这样，这不是SQL规范所要求的。此外，即使特定的DBMS总是按给出的GROUP BY子句排序数据，用户也可能会要求以不同的顺序排序。就因为你以某种方式分组数据（获得特定的分组聚集值），并不表示你需要以相同的方式排序输出。应该提供明确的ORDER BY子句，即使其效果等同于GROUP BY子句。

> 提示：不要忘记**ORDER BY**
> 一般在使用GROUP BY子句时，应该也给出ORDER BY子句。这是保证数据正确排序的唯一方法。千万不要仅依赖GROUP BY排序数据。

为说明GROUP BY和ORDER BY的使用方法，来看一个例子。下面的SELECT语句类似于前面那些例子。它检索包含三个或更多物品的订单号和订购物品的数目：

输入▼

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY order_num
HAVING COUNT(*) >= 3;
```

输出▼

```
order_num      items
---------      -----
20006          3
```

```
20007           5
20008           5
20009           3
```

要按订购物品的数目排序输出，需要添加ORDER BY子句，如下所示：

输入▼

```
SELECT order_num, COUNT(*) AS items
FROM OrderItems
GROUP BY order_num
HAVING COUNT(*) >= 3
ORDER BY items, order_num;
```

> 说明：**Access的不兼容性**
> Microsoft Access不允许按别名排序，因此这个例子在Access中将失败。解决方法是用实际的计算或字段位置替换items（在ORDER BY子句中），即ORDER BY COUNT(*), order_num或ORDER BY 2, order_num。

输出▼

```
order_num       items
---------       -----
20006           3
20009           3
20007           5
20008           5
```

分析▼

在这个例子中，使用GROUP BY子句按订单号（order_num列）分组数据，以便COUNT(*)函数能够返回每个订单中的物品数目。HAVING子句过滤数据，使得只返回包含三个或更多物品的订单。最后，用ORDER BY子句排序输出。

10.5 SELECT子句顺序

下面回顾一下SELECT语句中子句的顺序。表10-2以在SELECT语句中使用时必须遵循的次序，列出迄今为止所学过的子句。

表10-2 **SELECT**子句及其顺序

子　　句	说　　明	是否必须使用
SELECT	要返回的列或表达式	是
FROM	从中检索数据的表	仅在从表选择数据时使用
WHERE	行级过滤	否
GROUP BY	分组说明	仅在按组计算聚集时使用
HAVING	组级过滤	否
ORDER BY	输出排序顺序	否

10.6　小结

上一课介绍了如何用SQL聚集函数对数据进行汇总计算。这一课讲授了如何使用GROUP BY子句对多组数据进行汇总计算，返回每个组的结果。我们看到了如何使用HAVING子句过滤特定的组，还知道了ORDER BY和GROUP BY之间以及WHERE和HAVING之间的差异。

第11课　使用子查询

这一课介绍什么是子查询，如何使用它们。

11.1　子查询

SELECT语句是SQL的查询。我们迄今为止所看到的所有SELECT语句都是简单查询，即从单个数据库表中检索数据的单条语句。

> 查询（**query**）
> 任何SQL语句都是查询。但此术语一般指SELECT语句。

SQL还允许创建子查询（subquery），即嵌套在其他查询中的查询。为什么要这样做呢？理解这个概念的最好方法是考察几个例子。

> 说明：**MySQL**支持
> 如果使用MySQL，应该知道对子查询的支持是从4.1版本引入的。MySQL的早期版本不支持子查询。

11.2　利用子查询进行过滤

本书所有课中使用的数据库表都是关系表（关于每个表及关系的描述，请参阅附录A）。订单存储在两个表中。每个订单包含订单编号、客户ID、订单日期，在Orders表中存储为一行。各订单的物品存储在相关的OrderItems表中。Orders表不存储顾客信息，只存储顾客ID。顾客的实际信息存储在Customers表中。

现在，假如需要列出订购物品RGAN01的所有顾客，应该怎样检索？下面列出具体的步骤。

1. 检索包含物品RGAN01的所有订单的编号。
2. 检索具有前一步骤列出的订单编号的所有顾客的ID。
3. 检索前一步骤返回的所有顾客ID的顾客信息。

上述每个步骤都可以单独作为一个查询来执行。可以把一条SELECT语句返回的结果用于另一条SELECT语句的WHERE子句。

也可以使用子查询来把3个查询组合成一条语句。

第一条SELECT语句的含义很明确，它对prod_id为RGAN01的所有订单物品，检索其order_num列。输出列出了两个包含此物品的订单：

输入▼

```
SELECT order_num
FROM OrderItems
WHERE prod_id = 'RGAN01';
```

输出▼

```
order_num
-----------
20007
20008
```

现在，我们知道了哪个订单包含要检索的物品，下一步查询与订单20007和20008相关的顾客ID。利用第5课介绍的IN子句，编写如下的SELECT语句：

输入▼

```
SELECT cust_id
FROM Orders
WHERE orders_num IN (20007,20008);
```

输出▼

```
cust_id
----------
1000000004
1000000005
```

现在，结合这两个查询，把第一个查询（返回订单号的那一个）变为子查询。请看下面的SELECT语句：

输入▼

```
SELECT cust_id
FROM Orders
WHERE order_num IN (SELECT order_num
                    FROM OrderItems
                    WHERE prod_id = 'RGAN01');
```

输出▼

```
cust_id
----------
1000000004
1000000005
```

分析▼

在SELECT语句中，子查询总是从内向外处理。在处理上面的SELECT语句时，DBMS实际上执行了两个操作。

首先，它执行下面的查询：

```
SELECT order_num FROM orderitems WHERE prod_id='RGAN01'
```

此查询返回两个订单号：20007和20008。然后，这两个值以IN操作符要求的逗号分隔的格式传递给外部查询的WHERE子句。外部查询变成：

```
SELECT cust_id FROM orders WHERE order_num IN (20007,20008)
```

可以看到，输出是正确的，与前面硬编码WHERE子句所返回的值相同。

> **提示：格式化SQL**
> 包含子查询的SELECT语句难以阅读和调试，它们在较为复杂时更是如此。如上所示，把子查询分解为多行并进行适当的缩进，能极大地简化子查询的使用。
>
> 顺便一提，这就是颜色编码起作用的地方，好的DBMS客户端正是出于这个原因使用了颜色代码SQL。

现在得到了订购物品RGAN01的所有顾客的ID。下一步是检索这些顾客ID的顾客信息。检索两列的SQL语句为：

输入▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN ('1000000004','1000000005');
```

可以把其中的WHERE子句转换为子查询，而不是硬编码这些顾客ID：

输入▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE prod_id = 'RGAN01
```

输出▼

```
cust_name                        cust_contact
-----------------------------    --------------------
Fun4All                          Denise L. Stephens
The Toy Store                    Kim Howard
```

分析▼
为了执行上述SELECT语句，DBMS实际上必须执行三条SELECT语句。最里边的子查询返回订单号列表，此列表用于其外面的子查询的WHERE子句。外面的子查询返回顾客ID列表，此顾客ID列表用于最外层查询的WHERE子句。最外层查询返回所需的数据。

可见，在WHERE子句中使用子查询能够编写出功能很强且很灵活的SQL语句。对于能嵌套的子查询的数目没有限制，不过在实际使用时由于性能的限制，不能嵌套太多的子查询。

> 警告：只能是单列
> 作为子查询的SELECT语句只能查询单个列。企图检索多个列将返回错误。

> 警告：子查询和性能
> 这里给出的代码有效，并且获得了所需的结果。但是，使用子查询并不总是执行这类数据检索的最有效方法。更多的论述，请参阅第12课，其中将再次给出这个例子。

11.3　作为计算字段使用子查询

使用子查询的另一方法是创建计算字段。假如需要显示Customers表中每个顾客的订单总数。订单与相应的顾客ID存储在Orders表中。

执行这个操作，要遵循下面的步骤：

1. 从Customers表中检索顾客列表；
2. 对于检索出的每个顾客，统计其在Orders表中的订单数目。

正如前两课所述，可以使用SELECT COUNT(*)对表中的行进行计数，并且通过提供一条WHERE子句来过滤某个特定的顾客ID，仅对该顾客的订单进行计数。例如，下面的代码对顾客1000000001的订单进行计数：

输入▼

```
SELECT COUNT(*) AS orders
FROM Orders
WHERE cust_id = '1000000001';
```

要对每个顾客执行COUNT(*)，应该将它作为一个子查询。请看下面的代码：

输入▼

```
SELECT cust_name,
       cust_state,
       (SELECT COUNT(*)
        FROM Orders
        WHERE Orders.cust_id = Customers.cust_id) AS orders
FROM Customers
ORDER BY cust_name;
```

输出▼

```
cust_name                       cust_state    orders
-------------------------       ----------    ------
Fun4All                         IN            1
Fun4All                         AZ            1
Kids Place                      OH            0
The Toy Store                   IL            1
Village Toys                    MI            2
```

分析▼

这条SELECT语句对Customers表中每个顾客返回三列：cust_name、cust_state和orders。orders是一个计算字段，它是由圆括号中的子查询建立的。该子查询对检索出的每个顾客执行一次。在此例中，该子查询执行了5次，因为检索出了5个顾客。

子查询中的WHERE子句与前面使用的WHERE子句稍有不同，因为它使用了完全限定列名，而不只是列名（cust_id）。它指定表名和列名（Orders.cust_id和Customers.cust_id）。下面的WHERE子句告诉SQL，比较Orders表中的cust_id和当前正从Customers表中检索的cust_id：

```
WHERE Orders.cust_id = Customers.cust_id
```

用一个句点分隔表名和列名，在有可能混淆列名时必须使用这种语法。在这个例子中，有两个cust_id列：一个在Customers中，另一个在Orders中。如果不采用完全限定列名，DBMS会认为要对Orders表中的cust_id自身进行比较。因为

```
SELECT COUNT(*) FROM Orders WHERE cust_id = cust_id
```

总是返回Orders表中订单的总数，而这个结果不是我们想要的：

输入▼

```
SELECT cust_name,
       cust_state,
       (SELECT COUNT(*)
        FROM Orders
        WHERE cust_id = cust_id) AS orders
FROM Customers
ORDER BY cust_name;
```

输出▼

```
cust_name                      cust_state     orders
-------------------------      ----------     ------
Fun4All                        IN             5
Fun4All                        AZ             5
Kids Place                     OH             5
The Toy Store                  IL             5
Village Toys                   MI             5
```

虽然子查询在构造这种SELECT语句时极有用，但必须注意限制有歧义的列。

> 警告：完全限定列名
> 你已经看到了为什么要使用完全限定列名，没有具体指定就会返回错误结果，因为DBMS会误解你的意思。有时候，由于出现冲突列名而导致的歧义性，会引起DBMS抛出错误信息。例如，WHERE或ORDER BY子句指定的某个列名可能会出现在多个表中。好的做法是，如果在SELECT语句中操作多个表，就应使用完全限定列名来避免歧义。

> 提示：不只一种解决方案
> 正如这一课前面所述，虽然这里给出的样例代码运行良好，但它并不是解决这种数据检索的最有效方法。在后面两课学习JOIN时，我们还会遇到这个例子。

11.4　小结

这一课学习了什么是子查询，如何使用它们。子查询常用于WHERE子句的IN操作符中，以及用来填充计算列。我们举了这两种操作类型的例子。

第12课　联结表

这一课会介绍什么是联结，为什么使用联结，如何编写使用联结的SELECT语句。

12.1　联结

SQL最强大的功能之一就是能在数据查询的执行中联结（join）表。联结是利用SQL的SELECT能执行的最重要的操作，很好地理解联结及其语法是学习SQL的极为重要的部分。

在能够有效地使用联结前，必须了解关系表以及关系数据库设计的一些基础知识。下面的介绍并不能涵盖这一主题的所有内容，但作为入门已经够了。

12.1.1　关系表

理解关系表，最好是来看个例子。

有一个包含产品目录的数据库表，其中每类物品占一行。对于每一种物品，要存储的信息包括产品描述、价格，以及生产该产品的供应商。

现在有同一供应商生产的多种物品，那么在何处存储供应商名、地址、联系方法等供应商信息呢？将这些数据与产品信息分开存储的理由是：

- 同一供应商生产的每个产品，其供应商信息都是相同的，对每个产品重复此信息既浪费时间又浪费存储空间；
- 如果供应商信息发生变化，例如供应商迁址或电话号码变动，只需修改一次即可；
- 如果有重复数据（即每种产品都存储供应商信息），则很难保证每次输入该数据的方式都相同。不一致的数据在报表中就很难利用。

关键是，相同的数据出现多次决不是一件好事，这是关系数据库设计的基础。关系表的设计就是要把信息分解成多个表，一类数据一个表。各表通过某些共同的值互相关联（所以才叫关系数据库）。

在这个例子中可建立两个表：一个存储供应商信息，另一个存储产品信息。`Vendors`表包含所有供应商信息，每个供应商占一行，具有唯一的标识。此标识称为主键（primary key），可以是供应商ID或任何其他唯一值。

`Products`表只存储产品信息，除了存储供应商ID（`Vendors`表的主键）外，它不存储其他有关供应商的信息。`Vendors`表的主键将`Vendors`表与`Products`表关联，利用供应商ID能从`Vendors`表中找出相应供应商的详细信息。

这样做的好处是：

- 供应商信息不重复，不会浪费时间和空间；
- 如果供应商信息变动，可以只更新`Vendors`表中的单个记录，相关表中的数据不用改动；
- 由于数据不重复，数据显然是一致的，使得处理数据和生成报表更简单。

总之，关系数据可以有效地存储，方便地处理。因此，关系数据库的可伸缩性远比非关系数据库要好。

> 可伸缩（**scale**）
> 能够适应不断增加的工作量而不失败。设计良好的数据库或应用程序称为可伸缩性好（scale well）。

12.1.2　为什么使用联结

如前所述，将数据分解为多个表能更有效地存储，更方便地处理，并且可伸缩性更好。但这些好处是有代价的。

如果数据存储在多个表中，怎样用一条`SELECT`语句就检索出数据呢？

答案是使用联结。简单说，联结是一种机制，用来在一条`SELECT`语句中关联表，因此称为联结。使用特殊的语法，可以联结多个表返回一组输出，联结在运行时关联表中正确的行。

> 说明：使用交互式**DBMS**工具
> 重要的是，要理解联结不是物理实体。换句话说，它在实际的数据库表中并不存在。DBMS会根据需要建立联结，它在查询执行期间一直存在。

> 许多DBMS提供图形界面，用来交互式地定义表关系。这些工具极其有助于维护引用完整性。在使用关系表时，仅在关系列中插入合法数据是非常重要的。回到这里的例子，如果Products表中存储了无效的供应商ID，则相应的产品不可访问，因为它们没有关联到某个供应商。为避免这种情况发生，可指示数据库只允许在Products表的供应商ID列中出现合法值（即出现在Vendors表中的供应商）。引用完整性表示DBMS强制实施数据完整性规则。这些规则一般由提供了界面的DBMS管理。

12.2　创建联结

创建联结非常简单，指定要联结的所有表以及关联它们的方式即可。请看下面的例子：

输入▼

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products
WHERE Vendors.vend_id = Products.vend_id;
```

输出▼

```
vend_name                 prod_name                 prod_price
--------------------      --------------------      ----------
Doll House Inc.           Fish bean bag toy         3.4900
Doll House Inc.           Bird bean bag toy         3.4900
Doll House Inc.           Rabbit bean bag toy       3.4900
Bears R Us                8 inch teddy bear         5.9900
Bears R Us                12 inch teddy bear        8.9900
Bears R Us                18 inch teddy bear        11.9900
Doll House Inc.           Raggedy Ann               4.9900
Fun and Games             King doll                 9.4900
Fun and Games             Queen doll                9.4900
```

分析▼

我们来看这段代码。SELECT语句与前面所有语句一样指定要检索的列。这里最大的差别是所指定的两列（prod_name和prod_price）在一个表中，而第三列（vend_name）在另一个表中。

现在来看FROM子句。与以前的SELECT语句不一样，这条语句的FROM子句列出了两个表：Vendors和Products。它们就是这条SELECT语句联结的两个表的名字。这两个表用WHERE子句正确地联结，WHERE子句指示DBMS将Vendors表中的vend_id与Products表中的vend_id匹配起来。

可以看到，要匹配的两列指定为Vendors.vend_id和Products.vend_id。这里需要这种完全限定列名，如果只给出vend_id，DBMS就不知道指的是哪一个（每个表中有一个）。从前面的输出可以看到，一条SELECT语句返回了两个不同表中的数据。

> 警告：完全限定列名
> 就像前一课提到的，在引用的列可能出现歧义时，必须使用完全限定列名（用一个句点分隔表名和列名）。如果引用一个没有用表名限制的具有歧义的列名，大多数DBMS会返回错误。

12.2.1 WHERE子句的重要性

使用WHERE子句建立联结关系似乎有点奇怪，但实际上是有个很充分的理由的。要记住，在一条SELECT语句中联结几个表时，相应的关系是在运行中构造的。在数据库表的定义中没有指示DBMS如何对表进行联结的内容。你必须自己做这件事情。在联结两个表时，实际要做的是将第一个表中的每一行与第二个表中的每一行配对。WHERE子句作为过滤条件，只包含那些匹配给定条件（这里是联结条件）的行。没有WHERE子句，第一个表中的每一行将与第二个表中的每一行配对，而不管它们逻辑上是否能配在一起。

> 笛卡儿积（**cartesian product**）
> 由没有联结条件的表关系返回的结果为笛卡儿积。检索出的行的数目将是第一个表中的行数乘以第二个表中的行数。

理解这一点，请看下面的SELECT语句及其输出：

输入▼

```
SELECT vend_name, prod_name, prod_price
FROM Vendors, Products;
```

输出▼

```
vend_name              prod_name                           p
-----------------     ----------------------------       ------
Bears R Us               8 inch teddy bear                  5.9
Bears R Us               12 inch teddy bear                 8.9
Bears R Us               18 inch teddy bear                 11.
Bears R Us               Fish bean bag toy                  3.4
Bears R Us               Bird bean bag toy                  3.4
Bears R Us               Rabbit bean bag toy                3.4
Bears R Us            Raggedy Ann                        4.99
Bears R Us               King doll                          9.4
Bears R Us               Queen doll                         9.4
Bear Emporium         8 inch teddy bear                  5.99
Bear Emporium            12 inch teddy bear                 8.9
Bear Emporium            18 inch teddy bear                 11.
Bear Emporium            Fish bean bag toy                  3.4
Bear Emporium            Bird bean bag toy                  3.4
Bear Emporium            Rabbit bean bag toy                3.4
Bear Emporium            Raggedy Ann                        4.9
Bear Emporium            King doll                          9.4
Bear Emporium           Queen doll                         9.49
Doll House Inc.       8 inch teddy bear                  5.99
Doll House Inc.       12 inch teddy bear                 8.99
Doll House Inc.       18 inch teddy bear                 11.99
Doll House Inc.       Fish bean bag toy                  3.49
Doll House Inc.       Bird bean bag toy                  3.49
Doll House Inc.       Rabbit bean bag toy                3.49
Doll House Inc.       Raggedy Ann                        4.99
Doll House Inc.       King doll                          9.49
Doll House Inc.       Queen doll                         9.49
Furball Inc.            8 inch teddy bear                  5.99
Furball Inc.           12 inch teddy bear                 8.99
Furball Inc.           18 inch teddy bear                 11.99
Furball Inc.           Fish bean bag toy                  3.49
Furball Inc.           Bird bean bag toy                  3.49
Furball Inc.           Rabbit bean bag toy                3.49
Furball Inc.           Raggedy Ann                        4.99
Furball Inc.           King doll                          9.49
Furball Inc.           Queen doll                         9.49
Fun and Games         8 inch teddy bear                  5.99
Fun and Games           12 inch teddy bear                 8.99
Fun and Games           18 inch teddy bear                 11.9
Fun and Games           Fish bean bag toy                  3.49
Fun and Games           Bird bean bag toy                  3.49
Fun and Games           Rabbit bean bag toy                3.49
```

```
Fun and Games          Raggedy Ann              4.99
Fun and Games          King doll                9.49
Fun and Games          Queen doll               9.49
Jouets et ours       8 inch teddy bear         5.99
Jouets et ours       12 inch teddy bear        8.99
Jouets et ours       18 inch teddy bear        11.99
Jouets et ours       Fish bean bag toy         3.49
Jouets et ours       Bird bean bag toy         3.49
Jouets et ours       Rabbit bean bag toy       3.49
Jouets et ours       Raggedy Ann               4.99
Jouets et ours       King doll                 9.49
Jouets et ours       Queen doll                9.49
```

分析▼

从上面的输出可以看到，相应的笛卡儿积不是我们想要的。这里返回的数据用每个供应商匹配了每个产品，包括了供应商不正确的产品（即使供应商根本就没有产品）。

> 警告：不要忘了**WHERE**子句
> 要保证所有联结都有WHERE子句，否则DBMS将返回比想要的数据多得多的数据。同理，要保证WHERE子句的正确性。不正确的过滤条件会导致DBMS返回不正确的数据。

> 提示：叉联结
> 有时，返回笛卡儿积的联结，也称叉联结（cross join）。

12.2.2　内联结

目前为止使用的联结称为等值联结（equijoin），它基于两个表之间的相等测试。这种联结也称为内联结（inner join）。其实，可以对这种联结使用稍微不同的语法，明确指定联结的类型。下面的SELECT语句返回与前面例子完全相同的数据：

输入▼

```
SELECT vend_name, prod_name, prod_price
FROM Vendors INNER JOIN Products
 ON Vendors.vend_id = Products.vend_id;
```

分析▼

此语句中的SELECT与前面的SELECT语句相同，但FROM子句不同。这里，两个表之间的关系是以INNER JOIN指定的部分FROM子句。在使用这种语法时，联结条件用特定的ON子句而不是WHERE子句给出。传递给ON的实际条件与传递给WHERE的相同。

至于选用哪种语法，请参阅具体的DBMS文档。

> 说明：“正确的”语法
> ANSI SQL规范首选INNER JOIN语法，之前使用的是简单的等值语法。其实，SQL语言纯正论者是用鄙视的眼光看待简单语法的。这就是说，DBMS的确支持简单格式和标准格式，我建议你要理解这两种格式，具体使用就看你用哪个更顺手了。

12.2.3 联结多个表

SQL不限制一条SELECT语句中可以联结的表的数目。创建联结的基本规则也相同。首先列出所有表，然后定义表之间的关系。例如：

输入▼

```
SELECT prod_name, vend_name, prod_price, quantity
FROM OrderItems, Products, Vendors
WHERE Products.vend_id = Vendors.vend_id
 AND OrderItems.prod_id = Products.prod_id
 AND order_num = 20007;
```

输出▼

```
prod_name            vend_name          prod_price     quantity
---------------      -------------      ----------     --------
18 inch teddy bear   Bears R Us         11.9900        50
Fish bean bag toy    Doll House Inc.    3.4900         100
Bird bean bag toy    Doll House Inc.    3.4900         100
Rabbit bean bag toy  Doll House Inc.    3.4900         100
Raggedy Ann          Doll House Inc.    4.9900         50
```

分析▼

这个例子显示订单20007中的物品。订单物品存储在OrderItems表中。每个产品按其产品ID存储，它引用Products表中的产品。这些产品通过供应商ID联结到Vendors表中相应的供应商，供应商ID存储

在每个产品的记录中。这里的FROM子句列出三个表，WHERE子句定义这两个联结条件，而第三个联结条件用来过滤出订单20007中的物品。

> 警告：性能考虑
> DBMS在运行时关联指定的每个表，以处理联结。这种处理可能非常耗费资源，因此应该注意，不要联结不必要的表。联结的表越多，性能下降越厉害。

> 警告：联结中表的最大数目
> 虽然SQL本身不限制每个联结约束中表的数目，但实际上许多DBMS都有限制。请参阅具体的DBMS文档以了解其限制。

现在回顾一下第11课中的例子，如下的SELECT语句返回订购产品RGAN01的顾客列表：

输入▼

```
SELECT cust_name, cust_contact
FROM Customers
WHERE cust_id IN (SELECT cust_id
                  FROM Orders
                  WHERE order_num IN (SELECT order_num
                                      FROM OrderItems
                                      WHERE prod_id = 'RGAN01
```

如第11课所述，子查询并不总是执行复杂SELECT操作的最有效方法，下面是使用联结的相同查询：

输入▼

```
SELECT cust_name, cust_contact
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
 AND OrderItems.order_num = Orders.order_num
 AND prod_id = 'RGAN01';
```

输出▼

```
cust_name                               cust_contact
------------------------------          --------------------
Fun4All                                 Denise L. Stephens
The Toy Store                           Kim Howard
```

分析▼

如第11课所述，这个查询中的返回数据需要使用3个表。但在这里，我们没有在嵌套子查询中使用它们，而是使用了两个联结来连接表。这里有三个WHERE子句条件。前两个关联联结中的表，后一个过滤产品RGAN01的数据。

> 提示：多做实验
>
> 可以看到，执行任一给定的SQL操作一般不只一种方法。很少有绝对正确或绝对错误的方法。性能可能会受操作类型、所使用的DBMS、表中数据量、是否存在索引或键等条件的影响。因此，有必要试验不同的选择机制，找出最适合具体情况的方法。

12.3　小结

联结是SQL中一个最重要、最强大的特性，有效地使用联结需要对关系数据库设计有基本的了解。本课在介绍联结时，讲述了一些关系数据库设计的基本知识，包括等值联结（也称为内联结）这种最常用的联结。下一课将介绍如何创建其他类型的联结。

第13课　创建高级联结

本课讲解另外一些联结（包括它们的含义和使用方法），介绍如何使用表别名，如何对被联结的表使用聚集函数。

13.1　使用表别名

第7课介绍了如何使用别名引用被检索的表列。给列起别名的语法如下：

输入▼

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

SQL除了可以对列名和计算字段使用别名，还允许给表名起别名。这样做有两个主要理由：

- 缩短SQL语句；
- 允许在一条SELECT语句中多次使用相同的表。

请看下面的SELECT语句。它与前一课例子中所用的语句基本相同，但改成了使用别名：

输入▼

```
SELECT cust_name, cust_contact
FROM Customers AS C, Orders AS O, OrderItems AS OI
WHERE C.cust_id = O.cust_id
 AND OI.order_num = O.order_num
 AND prod_id = 'RGAN01';
```

分析▼
可以看到，FROM子句中的三个表全都有别名。Customers AS C使用C作为Customers的别名，如此等等。这样，就可以使用省略的C而不用全名Customers。在这个例子中，表别名只用于WHERE子

句。其实它不仅能用于WHERE子句，还可以用于SELECT的列表、ORDER BY子句以及其他语句部分。

> 警告：**Oracle中没有AS**
> Oracle不支持AS关键字。要在Oracle中使用别名，可以不用AS，简单地指定列名即可（因此，应该是Customers C，而不是Customers AS C）。

需要注意，表别名只在查询执行中使用。与列别名不一样，表别名不返回到客户端。

13.2　使用不同类型的联结

迄今为止，我们使用的只是内联结或等值联结的简单联结。现在来看三种其他联结：自联结（self-join）、自然联结（natural join）和外联结（outer join）。

13.2.1　自联结

如前所述，使用表别名的一个主要原因是能在一条SELECT语句中不只一次引用相同的表。下面举一个例子。

假如要给与Jim Jones同一公司的所有顾客发送一封信件。这个查询要求首先找出Jim Jones工作的公司，然后找出在该公司工作的顾客。下面是解决此问题的一种方法：

输入▼

```
SELECT cust_id, cust_name, cust_contact
FROM Customers
WHERE cust_name = (SELECT cust_name
                   FROM Customers
                   WHERE cust_contact = 'Jim Jones');
```

输出▼

```
cust_id      cust_name          cust_contact
---------    --------------     --------------
1000000003   Fun4All            Jim Jones
1000000004   Fun4All            Denise L. Stephens
```

分析▼

这是第一种解决方案，使用了子查询。内部的SELECT语句做了一个简单检索，返回Jim Jones工作公司的cust_name。该名字用于外部查询的WHERE子句中，以检索出为该公司工作的所有雇员（第11课中讲授了子查询，更多信息请参阅该课）。

现在来看使用联结的相同查询：

输入▼

```
SELECT c1.cust_id, c1.cust_name, c1.cust_contact
FROM Customers AS c1, Customers AS c2
WHERE c1.cust_name = c2.cust_name
 AND c2.cust_contact = 'Jim Jones';
```

输出▼

```
cust_id     cust_name       cust_contact
-------     -----------     --------------
1000000003  Fun4All         Jim Jones
1000000004  Fun4All         Denise L. Stephens
```

> 提示：**Oracle中没有AS**
> Oracle用户应该记住去掉AS。

分析▼

此查询中需要的两个表实际上是相同的表，因此Customers表在FROM子句中出现了两次。虽然这是完全合法的，但对Customers的引用具有歧义性，因为DBMS不知道你引用的是哪个Customers表。

解决此问题，需要使用表别名。Customers第一次出现用了别名c1，第二次出现用了别名c2。现在可以将这些别名用作表名。例如，SELECT语句使用c1前缀明确给出所需列的全名。如果不这样，DBMS将返回错误，因为名为cust_id、cust_name、cust_contact的列各有两个。DBMS不知道想要的是哪一列（即使它们其实是同一列）。WHERE首先联结两个表，然后按第二个表中的cust_contact过滤数据，返回所需的数据。

> 提示：用自联结而不用子查询
> 自联结通常作为外部语句，用来替代从相同表中检索数据的使用子查询语句。虽然最终的结果是相同的，但许多DBMS处理联结

> 远比处理子查询快得多。应该试一下两种方法，以确定哪一种的性能更好。

13.2.2　自然联结

无论何时对表进行联结，应该至少有一列不只出现在一个表中（被联结的列）。标准的联结（前一课中介绍的内联结）返回所有数据，相同的列甚至多次出现。自然联结排除多次出现，使每一列只返回一次。

怎样完成这项工作呢？答案是，系统不完成这项工作，由你自己完成它。自然联结要求你只能选择那些唯一的列，一般通过对一个表使用通配符（SELECT *），而对其他表的列使用明确的子集来完成。下面举一个例子：

输入▼

```
SELECT C.*, O.order_num, O.order_date,
       OI.prod_id, OI.quantity, OI.item_price
FROM Customers AS C, Orders AS O, OrderItems AS OI
WHERE C.cust_id = O.cust_id
 AND OI.order_num = O.order_num
 AND prod_id = 'RGAN01';
```

> 提示：**Oracle中没有AS**
> Oracle用户应该记住去掉AS。

分析▼
在这个例子中，通配符只对第一个表使用。所有其他列明确列出，所以没有重复的列被检索出来。

事实上，我们迄今为止建立的每个内联结都是自然联结，很可能永远都不会用到不是自然联结的内联结。

13.2.3　外联结

许多联结将一个表中的行与另一个表中的行相关联，但有时候需要包含没有关联行的那些行。例如，可能需要使用联结完成以下工作：

- 对每个顾客下的订单进行计数，包括那些至今尚未下订单的顾

客；

- 列出所有产品以及订购数量，包括没有人订购的产品；
- 计算平均销售规模，包括那些至今尚未下订单的顾客。

在上述例子中，联结包含了那些在相关表中没有关联行的行。这种联结称为外联结。

> 注意：语法差别
> 需要注意，用来创建外联结的语法在不同的SQL实现中可能稍有不同。下面段落中描述的各种语法形式覆盖了大多数实现，在继续学习之前请参阅你使用的DBMS文档，以确定其语法。

下面的SELECT语句给出了一个简单的内联结。它检索所有顾客及其订单：

输入▼

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers INNER JOIN Orders
 ON Customers.cust_id = Orders.cust_id;
```

外联结语法类似。要检索包括没有订单顾客在内的所有顾客，可如下进行：

输入▼

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers LEFT OUTER JOIN Orders
 ON Customers.cust_id = Orders.cust_id;
```

输出▼

```
cust_id         order_num
----------      ---------
1000000001      20005
1000000001      20009
1000000002      NULL
1000000003      20006
1000000004      20007
1000000005      20008
```

分析▼

类似上一课提到的内联结，这条SELECT语句使用了关键字OUTER JOIN来指定联结类型（而不是在WHERE子句中指定）。但是，与内联结关联两个表中的行不同的是，外联结还包括没有关联行的行。在使用OUTER JOIN语法时，必须使用RIGHT或LEFT关键字指定包括其所有行的表（RIGHT指出的是OUTER JOIN右边的表，而LEFT指出的是OUTER JOIN左边的表）。上面的例子使用LEFT OUTER JOIN从FROM子句左边的表（Customers表）中选择所有行。为了从右边的表中选择所有行，需要使用RIGHT OUTER JOIN，如下例所示：

输入▼

```
SELECT Customers.cust_id, Orders.order_num
FROM Customers RIGHT OUTER JOIN Orders
 ON Orders.cust_id = Customers.cust_id;
```

> 注意：**SQLite外联结**
> SQLite支持LEFT OUTER JOIN，但不支持RIGHT OUTER JOIN。幸好，如果你确实需要在SQLite中使用RIGHT OUTER JOIN，有一种更简单的办法，这将在下面的提示中介绍。

> 提示：外联结的类型
> 要记住，总是有两种基本的外联结形式：左外联结和右外联结。它们之间的唯一差别是所关联的表的顺序。换句话说，调整FROM或WHERE子句中表的顺序，左外联结可以转换为右外联结。因此，这两种外联结可以互换使用，哪个方便就用哪个。

还存在另一种外联结，就是全外联结（full outer join），它检索两个表中的所有行并关联那些可以关联的行。与左外联结或右外联结包含一个表的不关联的行不同，全外联结包含两个表的不关联的行。全外联结的语法如下：

输入▼

```
SELECT Customers.cust_id, Orders.order_num
FROM Orders FULL OUTER JOIN Customers
ON Orders.cust_id = Customers.cust_id;
```

> 注意：**FULL OUTER JOIN**的支持
> Access、MariaDB、MySQL、Open Office Base和SQLite不支持FULL OUTER JOIN语法。

13.3 使用带聚集函数的联结

如第9课所述，聚集函数用来汇总数据。虽然至今为止我们举的聚集函数的例子都只是从一个表中汇总数据，但这些函数也可以与联结一起使用。

我们来看个例子，要检索所有顾客及每个顾客所下的订单数，下面的代码使用COUNT()函数完成此工作：

输入▼

```
SELECT Customers.cust_id,
       COUNT(Orders.order_num) AS num_ord
FROM Customers INNER JOIN Orders
 ON Customers.cust_id = Orders.cust_id
GROUP BY Customers.cust_id;
```

输出▼

```
cust_id         um_ord
----------      --------
1000000001      2
1000000003      1
1000000004      1
1000000005      1
```

分析▼
这条SELECT语句使用INNER JOIN将Customers和Orders表互相关联。GROUP BY子句按顾客分组数据，因此，函数调用COUNT(Orders.order_num)对每个顾客的订单计数，将它作为num_ord返回。

聚集函数也可以方便地与其他联结一起使用。请看下面的例子：

输入▼

```
SELECT Customers.cust_id,
```

```
       COUNT(Orders.order_num) AS num_ord
FROM Customers LEFT OUTER JOIN Orders
 ON Customers.cust_id = Orders.cust_id
GROUP BY Customers.cust_id;
```

> 提示：**Oracle中没有AS**
> 再次提醒Oracle用户，请记住删除AS。

输出▼

```
cust_id          num_ord
----------       --------
1000000001       2
1000000002       0
1000000003       1
1000000004       1
1000000005       1
```

分析▼
这个例子使用左外部联结来包含所有顾客，甚至包含那些没有任何订单的顾客。结果中也包含了顾客1000000002，他有0个订单。

13.4　使用联结和联结条件

在总结讨论联结的这两课前，有必要汇总一下联结及其使用的要点。

- 注意所使用的联结类型。一般我们使用内联结，但使用外联结也有效。
- 关于确切的联结语法，应该查看具体的文档，看相应的DBMS支持何种语法（大多数DBMS使用这两课中描述的某种语法）。
- 保证使用正确的联结条件（不管采用哪种语法），否则会返回不正确的数据。
- 应该总是提供联结条件，否则会得出笛卡儿积。
- 在一个联结中可以包含多个表，甚至可以对每个联结采用不同的联结类型。虽然这样做是合法的，一般也很有用，但应该在一起测试它们前分别测试每个联结。这会使故障排除更为简单。

13.5　小结

本课是上一课的延续，首先讲授了如何以及为什么使用别名，然后讨论不同的联结类型以及每类联结所使用的语法。我们还介绍了如何与联结一起使用聚集函数，以及在使用联结时应该注意的问题。

第14课　组合查询

本课讲述如何利用UNION操作符将多条SELECT语句组合成一个结果集。

14.1　组合查询

多数SQL查询只包含从一个或多个表中返回数据的单条SELECT语句。但是，SQL也允许执行多个查询（多条SELECT语句），并将结果作为一个查询结果集返回。这些组合查询通常称为并（union）或复合查询（compound query）。

主要有两种情况需要使用组合查询：

- 在一个查询中从不同的表返回结构数据；
- 对一个表执行多个查询，按一个查询返回数据。

> 提示：组合查询和多个**WHERE**条件
> 多数情况下，组合相同表的两个查询所完成的工作与具有多个WHERE子句条件的一个查询所完成的工作相同。换句话说，任何具有多个WHERE子句的SELECT语句都可以作为一个组合查询，在下面可以看到这一点。

14.2　创建组合查询

可用UNION操作符来组合数条SQL查询。利用UNION，可给出多条SELECT语句，将它们的结果组合成一个结果集。

14.2.1　使用UNION

使用UNION很简单，所要做的只是给出每条SELECT语句，在各条语句之间放上关键字UNION。

举个例子，假如需要Illinois、Indiana和Michigan等美国几个州的所有顾客的报表，还想包括不管位于哪个州的所有的Fun4All。当然可以利用WHERE子句来完成此工作，不过这次我们使用UNION。

如上所述，创建UNION涉及编写多条SELECT语句。首先来看单条语句：

输入▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI');
```

输出▼

```
cust_name          cust_contact         cust_email
-----------        -------------        ------------
Village Toys       John Smith           sales@villagetoys.com
Fun4All            Jim Jones            jjones@fun4all.com
The Toy Store      Kim Howard           NULL
```

输入▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

输出▼

```
cust_name          cust_contact            cust_email
-----------        -------------           ------------
Fun4All            Jim Jones               jjones@fun4all.com
Fun4All            Denise L. Stephens      dstephens@fun4all.com
```

分析▼

第一条SELECT把Illinois、Indiana、Michigan等州的缩写传递给IN子句，检索出这些州的所有行。第二条SELECT利用简单的相等测试找出所有Fun4All。

组合这两条语句，可以如下进行：

输入▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
```

```
WHERE cust_state IN ('IL','IN','MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

输出▼

```
cust_name          cust_contact         cust_email
-----------        -----------          ----------------
Fun4All            Denise L. Stephens   dstephens@fun4all.com
Fun4All            Jim Jones            jjones@fun4all.com
Village Toys       John Smith           sales@villagetoys.com
The Toy Store      Kim Howard           NULL
```

分析▼
这条语句由前面的两条SELECT语句组成，之间用UNION关键字分隔。UNION指示DBMS执行这两条SELECT语句，并把输出组合成一个查询结果集。

为了便于参考，这里给出使用多条WHERE子句而不是UNION的相同查询：

输入▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI')
 OR cust_name = 'Fun4All';
```

在这个简单的例子中，使用UNION可能比使用WHERE子句更为复杂。但对于较复杂的过滤条件，或者从多个表（而不是一个表）中检索数据的情形，使用UNION可能会使处理更简单。

> 提示：**UNION**的限制
> 使用UNION组合SELECT语句的数目，SQL没有标准限制。但是，最好是参考一下具体的DBMS文档，了解它是否对UNION能组合的最大语句数目有限制。

> 警告：性能问题
> 多数好的DBMS使用内部查询优化程序，在处理各条SELECT语句前组合它们。理论上讲，这意味着从性能上看使用多条WHERE子句条件还是UNION应该没有实际的差别。不过我说的是理论上，实践中多数查询优化程序并不能达到理想状态，所以最好测试一下这两种方法，看哪种工作得更好。

14.2.2　UNION规则

可以看到，UNION非常容易使用，但在进行组合时需要注意几条规则。

- UNION必须由两条或两条以上的SELECT语句组成，语句之间用关键字UNION分隔（因此，如果组合四条SELECT语句，将要使用三个UNION关键字）。
- UNION中的每个查询必须包含相同的列、表达式或聚集函数（不过，各个列不需要以相同的次序列出）。
- 列数据类型必须兼容：类型不必完全相同，但必须是DBMS可以隐含转换的类型（例如，不同的数值类型或不同的日期类型）。

如果遵守了这些基本规则或限制，则可以将UNION用于任何数据检索操作。

14.2.3　包含或取消重复的行

回到14.2.1节，我们看看所用的SELECT语句。注意到在分别执行语句时，第一条SELECT语句返回3行，第二条SELECT语句返回2行。而在用UNION组合两条SELECT语句后，只返回4行而不是5行。

UNION从查询结果集中自动去除了重复的行；换句话说，它的行为与一条SELECT语句中使用多个WHERE子句条件一样。因为Indiana州有一个Fun4All单位，所以两条SELECT语句都返回该行。使用UNION时，重复的行会被自动取消。

这是UNION的默认行为，如果愿意也可以改变它。事实上，如果想返回所有的匹配行，可使用UNION ALL而不是UNION。

请看下面的例子：

输入▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI')
UNION ALL
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All';
```

输出▼

```
cust_name        cust_contact          cust_email
-----------      -------------         ------------
Village Toys     John Smith            sales@villagetoys.com
Fun4All          Jim Jones             jjones@fun4all.com
The Toy Store    Kim Howard            NULL
Fun4All          Jim Jones             jjones@fun4all.com
Fun4All          Denise L. Stephens    dstephens@fun4all.com
```

分析▼

使用UNION ALL，DBMS不取消重复的行。因此，这里返回5行，其中有一行出现两次。

> 提示：**UNION与WHERE**
> 这一课一开始我们说过，UNION几乎总是完成与多个WHERE条件相同的工作。UNION ALL为UNION的一种形式，它完成WHERE子句完成不了的工作。如果确实需要每个条件的匹配行全部出现（包括重复行），就必须使用UNION ALL，而不是WHERE。

14.2.4　对组合查询结果排序

SELECT语句的输出用ORDER BY子句排序。在用UNION组合查询时，只能使用一条ORDER BY子句，它必须位于最后一条SELECT语句之后。对于结果集，不存在用一种方式排序一部分，而又用另一种方式排序另一部分的情况，因此不允许使用多条ORDER BY子句。

下面的例子对前面UNION返回的结果进行排序：

输入▼

```
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_state IN ('IL','IN','MI')
UNION
SELECT cust_name, cust_contact, cust_email
FROM Customers
WHERE cust_name = 'Fun4All'
ORDER BY cust_name, cust_contact;
```

输出▼

```
cust_name          cust_contact          cust_email
-----------        -------------         -------------
Fun4All            Denise L. Stephens    dstephens@fun4all.com
Fun4All            Jim Jones             jjones@fun4all.com
The Toy Store      Kim Howard            NULL
Village Toys       John Smith            sales@villagetoys.com
```

分析▼

这条UNION在最后一条SELECT语句后使用了ORDER BY子句。虽然ORDER BY子句似乎只是最后一条SELECT语句的组成部分，但实际上DBMS将用它来排序所有SELECT语句返回的所有结果。

> 说明：其他类型的**UNION**
> 某些DBMS还支持另外两种UNION：EXCEPT（有时称为MINUS）可用来检索只在第一个表中存在而在第二个表中不存在的行；而INTERSECT可用来检索两个表中都存在的行。实际上，这些UNION很少使用，因为相同的结果可利用联结得到。

> 提示：操作多个表
> 为了简单，本课中的例子都是使用UNION来组合针对同一表的多个查询。实际上，UNION在需要组合多个表的数据时也很有用，即使是有不匹配列名的表，在这种情况下，可以将UNION与别名组合，检索一个结果集。

14.3　小结

这一课讲授如何用UNION操作符来组合SELECT语句。利用UNION，

可以把多条查询的结果作为一条组合查询返回，不管结果中有无重复。使用UNION可极大地简化复杂的WHERE子句，简化从多个表中检索数据的工作。

第15课　插入数据

这一课介绍如何利用SQL的INSERT语句将数据插入表中。

15.1　数据插入

毫无疑问，SELECT是最常用的SQL语句了，这就是前14课都在讲它的原因。但是，还有其他3个常用的SQL语句需要学习。第一个就是INSERT（下一课介绍另外两个）。

顾名思义，INSERT用来将行插入（或添加）到数据库表。插入有几种方式：

- 插入完整的行；
- 插入行的一部分；
- 插入某些查询的结果。

下面逐一介绍这些内容。

> 提示：插入及系统安全
> 使用INSERT语句可能需要客户端/服务器DBMS中的特定安全权限。在你试图使用INSERT前，应该保证自己有足够的安全权限。

15.1.1　插入完整的行

把数据插入表中的最简单方法是使用基本的INSERT语法，它要求指定表名和插入到新行中的值。下面举一个例子：

输入▼

```
INSERT INTO Customers
VALUES('1000000006',
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111',
```

```
        'USA',
        NULL,
        NULL);
```

分析▼ 这个例子将一个新顾客插入到Customers表中。存储到表中每一列的数据在VALUES子句中给出，必须给每一列提供一个值。如果某列没有值，如上面的cust_contact和cust_email列，则应该使用NULL值（假定表允许对该列指定空值）。各列必须以它们在表定义中出现的次序填充。

> 提示：**INTO**关键字
> 在某些SQL实现中，跟在INSERT之后的INTO关键字是可选的。但是，即使不一定需要，最好还是提供这个关键字，这样做将保证SQL代码在DBMS之间可移植。

虽然这种语法很简单，但并不安全，应该尽量避免使用。上面的SQL语句高度依赖于表中列的定义次序，还依赖于其容易获得的次序信息。即使可以得到这种次序信息，也不能保证各列在下一次表结构变动后保持完全相同的次序。因此，编写依赖于特定列次序的SQL语句是很不安全的，这样做迟早会出问题。

编写INSERT语句的更安全（不过更烦琐）的方法如下：

输入▼

```
INSERT INTO Customers(cust_id,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
                      cust_country,
                      cust_contact,
                      cust_email)
VALUES('1000000006',
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111',
       'USA',
       NULL,
```

```
        NULL);
```

分析▾

这个例子与前一个INSERT语句的工作完全相同，但在表名后的括号里明确给出了列名。在插入行时，DBMS将用VALUES列表中的相应值填入列表中的对应项。VALUES中的第一个值对应于第一个指定列名，第二个值对应于第二个列名，如此等等。

因为提供了列名，VALUES必须以其指定的次序匹配指定的列名，不一定按各列出现在表中的实际次序。其优点是，即使表的结构改变，这条INSERT语句仍然能正确工作。

下面的INSERT语句填充所有列（与前面的一样），但以一种不同的次序填充。因为给出了列名，所以插入结果仍然正确：

输入▾

```
INSERT INTO Customers(cust_id,
                      cust_contact,
                      cust_email,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip)
VALUES('1000000006',
       NULL,
       NULL,
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111');
```

> 提示：总是使用列的列表
> 不要使用没有明确给出列的INSERT语句。给出列能使SQL代码继续发挥作用，即使表结构发生了变化。

> 警告：小心使用**VALUES**
> 不管使用哪种INSERT语法，VALUES的数目都必须正确。如果不

提供列名，则必须给每个表列提供一个值；如果提供列名，则必须给列出的每个列一个值。否则，就会产生一条错误消息，相应的行不能成功插入。

15.1.2　插入部分行

正如所述，使用INSERT的推荐方法是明确给出表的列名。使用这种语法，还可以省略列，这表示可以只给某些列提供值，给其他列不提供值。

请看下面的例子：

输入▼

```
INSERT INTO Customers(cust_id,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
                      cust_country)
VALUES('1000000006',
       'Toy Land',
       '123 Any Street',
       'New York',
       'NY',
       '11111',
       'USA');
```

分析▼

在本课前面的例子中，没有给cust_contact和cust_email这两列提供值。这表示没必要在INSERT语句中包含它们。因此，这里的INSERT语句省略了这两列及其对应的值。

> 警告：省略列
>
> 如果表的定义允许，则可以在INSERT操作中省略某些列。省略的列必须满足以下某个条件。
>
> - 该列定义为允许NULL值（无值或空值）。
> - 在表定义中给出默认值。这表示如果不给出值，将使用默认值。

如果对表中不允许NULL值且没有默认值的列不给出值，DBMS将产生错误消息，并且相应的行插入不成功。

警告：省略所需的值
如果表中不允许有NULL值或者默认值，这时却省略了表中的值，DBMS就会产生错误消息，相应的行不能成功插入。

15.1.3　插入检索出的数据

INSERT一般用来给表插入具有指定列值的行。INSERT还存在另一种形式，可以利用它将SELECT语句的结果插入表中，这就是所谓的INSERT SELECT。顾名思义，它是由一条INSERT语句和一条SELECT语句组成的。

假如想把另一表中的顾客列合并到Customers表中。不需要每次读取一行再将它用INSERT插入，可以如下进行：

输入▼

```
INSERT INTO Customers(cust_id,
                      cust_contact,
                      cust_email,
                      cust_name,
                      cust_address,
                      cust_city,
                      cust_state,
                      cust_zip,
                      cust_country)
SELECT cust_id,
       cust_contact,
       cust_email,
       cust_name,
       cust_address,
       cust_city,
       cust_state,
       cust_zip,
       cust_country
FROM CustNew;
```

> 说明：新例子的说明
> 这个例子从一个名为CustNew的表中读出数据并插入到Customers表。为了试验这个例子，应该首先创建和填充CustNew表。CustNew表的结构与附录A中描述的Customers表相同。在填充CustNew时，不应该使用已经在Customers中用过的cust_id值（如果主键值重复，后续的INSERT操作将会失败）。

分析▼

这个例子使用INSERT SELECT从CustNew中将所有数据导入Customers。SELECT语句从CustNew检索出要插入的值，而不是列出它们。SELECT中列出的每一列对应于Customers表名后所跟的每一列。这条语句将插入多少行呢？这依赖于CustNew表有多少行。如果这个表为空，则没有行被插入（也不产生错误，因为操作仍然是合法的）。如果这个表确实有数据，则所有数据将被插入到Customers。

> 提示：**INSERT SELECT**中的列名
> 为简单起见，这个例子在INSERT和SELECT语句中使用了相同的列名。但是，不一定要求列名匹配。事实上，DBMS一点儿也不关心SELECT返回的列名。它使用的是列的位置，因此SELECT中的第一列（不管其列名）将用来填充表列中指定的第一列，第二列将用来填充表列中指定的第二列，如此等等。

INSERT SELECT中SELECT语句可以包含WHERE子句，以过滤插入的数据。

> 提示：插入多行
> INSERT通常只插入一行。要插入多行，必须执行多个INSERT语句。INSERT SELECT是个例外，它可以用一条INSERT插入多行，不管SELECT语句返回多少行，都将被INSERT插入。

15.2　从一个表复制到另一个表

有一种数据插入不使用INSERT语句。要将一个表的内容复制到一个全新的表（运行中创建的表），可以使用SELECT INTO语句。

> 说明：**DB2**不支持
> DB2不支持这里描述的SELECT INTO。

与INSERT SELECT将数据添加到一个已经存在的表不同，SELECT INTO将数据复制到一个新表（有的DBMS可以覆盖已经存在的表，这依赖于所使用的具体DBMS）。

> 说明：**INSERT SELECT与SELECT INTO**
> 它们之间的一个重要差别是前者插入数据，而后者导出数据。

下面的例子说明如何使用SELECT INTO：

输入▼

```
SELECT *
INTO CustCopy
FROM Customers;
```

分析▼
这条SELECT语句创建一个名为CustCopy的新表，并把Customers表的整个内容复制到新表中。因为这里使用的是SELECT *，所以将在CustCopy表中创建（并填充）与Customers表的每一列相同的列。要想只复制部分的列，可以明确给出列名，而不是使用*通配符。

MariaDB、MySQL、Oracle、PostgreSQL和SQLite使用的语法稍有不同：

输入▼

```
CREATE TABLE CustCopy AS
SELECT * FROM Customers;
```

在使用SELECT INTO时，需要知道一些事情：

- 任何SELECT选项和子句都可以使用，包括WHERE和GROUP BY；
- 可利用联结从多个表插入数据；
- 不管从多少个表中检索数据，数据都只能插入到一个表中。

> 提示：进行表的复制
> 先用SELECT INTO复制表，再试验新SQL语句，是很好的做法。先进行复制，可在复制的数据上测试SQL代码，而不会影响实际的数据。

> 说明：更多例子
> 如果想看INSERT用法的更多例子，请参阅附录A中给出的样例表填充脚本。

15.3　小结

这一课介绍如何将行插入到数据库表中。我们学习了使用INSERT的几种方法，为什么要明确使用列名，如何用INSERT SELECT从其他表中导入行，如何用SELECT INTO将行导出到一个新表。下一课将讲述如何使用UPDATE和DELETE进一步操作表数据。

第16课　更新和删除数据

这一课介绍如何利用UPDATE和DELETE语句进一步操作表数据。

16.1　更新数据

更新（修改）表中的数据，可以使用UPDATE语句。有两种使用UPDATE的方式：

- 更新表中的特定行；
- 更新表中的所有行。

下面分别介绍。

> 警告：不要省略**WHERE**子句
> 在使用UPDATE时一定要细心。因为稍不注意，就会更新表中的所有行。使用这条语句前，请完整地阅读本节。

> 提示：**UPDATE**与安全
> 在客户端/服务器的DBMS中，使用UPDATE语句可能需要特殊的安全权限。在你使用UPDATE前，应该保证自己有足够的安全权限。

使用UPDATE语句非常容易，甚至可以说太容易了。基本的UPDATE语句由三部分组成，分别是：

- 要更新的表；
- 列名和它们的新值；
- 确定要更新哪些行的过滤条件。

举一个简单例子。客户1000000005现在有了电子邮件地址，因此他的记录需要更新，语句如下：

输入▼

```
UPDATE Customers
```

```
SET cust_email = 'kim@thetoystore.com'
WHERE cust_id = '1000000005';
```

UPDATE语句总是以要更新的表名开始。在这个例子中，要更新的表名为Customers。SET命令用来将新值赋给被更新的列。在这里，SET子句设置cust_email列为指定的值：

```
SET cust_email = 'kim@thetoystore.com'
```

UPDATE语句以WHERE子句结束，它告诉DBMS更新哪一行。没有WHERE子句，DBMS将会用这个电子邮件地址更新Customers表中的所有行，这不是我们希望的。

更新多个列的语法稍有不同：

输入▼

```
UPDATE Customers
SET cust_contact = 'Sam Roberts',
    cust_email = 'sam@toyland.com'
WHERE cust_id = '1000000006';
```

在更新多个列时，只需要使用一条SET命令，每个“列＝值”对之间用逗号分隔（最后一列之后不用逗号）。在此例子中，更新顾客1000000006的cust_contact和cust_email列。

> 提示：在**UPDATE**语句中使用子查询
> UPDATE语句中可以使用子查询，使得能用SELECT语句检索出的数据更新列数据。关于子查询及使用的更多内容，请参阅第11课。

> 提示：**FROM**关键字
> 有的SQL实现支持在UPDATE语句中使用FROM子句，用一个表的数据更新另一个表的行。如想知道你的DBMS是否支持这个特性，请参阅它的文档。

要删除某个列的值，可设置它为NULL（假如表定义允许NULL值）。如下进行：

输入▼

```
UPDATE Customers
SET cust_email = NULL
WHERE cust_id = '1000000005';
```

其中NULL用来去除cust_email列中的值。这与保存空字符串很不同（空字符串用''表示，是一个值），而NULL表示没有值。

16.2　删除数据

从一个表中删除（去掉）数据，使用DELETE语句。有两种使用DELETE的方式：

- 从表中删除特定的行；
- 从表中删除所有行。

下面分别介绍。

> 警告：不要省略**WHERE**子句
> 在使用DELETE时一定要细心。因为稍不注意，就会错误地删除表中所有行。在使用这条语句前，请完整地阅读本节。

> 提示：**DELETE**与安全
> 在客户端/服务器的DBMS中，使用DELETE语句可能需要特殊的安全权限。在你使用DELETE前，应该保证自己有足够的安全权限。

前面说过，UPDATE非常容易使用，而DELETE更容易使用。

下面的语句从Customers表中删除一行：

输入▼

```
DELETE FROM Customers
WHERE cust_id = '1000000006';
```

这条语句很容易理解。DELETE FROM要求指定从中删除数据的表

名，WHERE子句过滤要删除的行。在这个例子中，只删除顾客1000000006。如果省略WHERE子句，它将删除表中每个顾客。

> 提示：友好的外键
> 第12课介绍了联结，简单联结两个表只需要这两个表中的公用字段。也可以让DBMS通过使用外键来严格实施关系（这些定义在附录A中）。存在外键时，DBMS使用它们实施引用完整性。例如要向Products表中插入一个新产品，DBMS不允许通过未知的供应商id插入它，因为vend_id列是作为外键连接到Vendors表的。那么，这与DELETE有什么关系呢？使用外键确保引用完整性的一个好处是，DBMS通常可以防止删除某个关系需要用到的行。例如，要从Products表中删除一个产品，而这个产品用在OrderItems的已有订单中，那么DELETE语句将抛出错误并中止。这是总要定义外键的另一个理由。

> 提示：**FROM**关键字
> 在某些SQL实现中，跟在DELETE后的关键字FROM是可选的。但是即使不需要，也最好提供这个关键字。这样做将保证SQL代码在DBMS之间可移植。

DELETE不需要列名或通配符。DELETE删除整行而不是删除列。要删除指定的列，请使用UPDATE语句。

> 说明：删除表的内容而不是表
> DELETE语句从表中删除行，甚至是删除表中所有行。但是，DELETE不删除表本身。

> 提示：更快的删除
> 如果想从表中删除所有行，不要使用DELETE。可使用TRUNCATE TABLE语句，它完成相同的工作，而速度更快（因为不记录数据的变动）。

16.3　更新和删除的指导原则

前两节使用的UPDATE和DELETE语句都有WHERE子句，这样做的理由很充分。如果省略了WHERE子句，则UPDATE或DELETE将被应用到表中所有的行。换句话说，如果执行UPDATE而不带WHERE子句，则表中每一行都将用新值更新。类似地，如果执行DELETE语句而不带WHERE子句，表的所有数据都将被删除。

下面是许多SQL程序员使用UPDATE或DELETE时所遵循的重要原则。

- 除非确实打算更新和删除每一行，否则绝对不要使用不带WHERE子句的UPDATE或DELETE语句。
- 保证每个表都有主键（如果忘记这个内容，请参阅第12课），尽可能像WHERE子句那样使用它（可以指定各主键、多个值或值的范围）。
- 在UPDATE或DELETE语句使用WHERE子句前，应该先用SELECT进行测试，保证它过滤的是正确的记录，以防编写的WHERE子句不正确。
- 使用强制实施引用完整性的数据库（关于这个内容，请参阅第12课），这样DBMS将不允许删除其数据与其他表相关联的行。
- 有的DBMS允许数据库管理员施加约束，防止执行不带WHERE子句的UPDATE或DELETE语句。如果所采用的DBMS支持这个特性，应该使用它。

若是SQL没有撤销（undo）按钮，应该非常小心地使用UPDATE和DELETE，否则你会发现自己更新或删除了错误的数据。

16.4　小结

这一课讲述了如何使用UPDATE和DELETE语句处理表中的数据。我们学习了这些语句的语法，知道了它们可能存在的危险，了解了为什么WHERE子句对UPDATE和DELETE语句很重要，还学习了为保证数据安全而应该遵循的一些指导原则。

第17课　创建和操纵表

这一课讲授创建、更改和删除表的基本知识。

17.1　创建表

SQL不仅用于表数据操纵，还用来执行数据库和表的所有操作，包括表本身的创建和处理。

一般有两种创建表的方法：

- 多数DBMS都具有交互式创建和管理数据库表的工具；
- 表也可以直接用SQL语句操纵。

用程序创建表，可以使用SQL的`CREATE TABLE`语句。需要注意的是，使用交互式工具时实际上就是使用SQL语句。这些语句不是用户编写的，界面工具会自动生成并执行相应的SQL语句（更改已有的表时也是这样）。

> 警告：语法差别
> 在不同的SQL实现中，`CREATE TABLE`语句的语法可能有所不同。对于具体的DBMS支持何种语法，请参阅相应的文档。

这一课不会介绍创建表时可以使用的所有选项，那超出了本课的范围，我只给出一些基本选项。详细的信息说明，请参阅具体的DBMS文档。

> 说明：**DBMS**创建表的具体例子
> 关于DBMS的`CREATE TABLE`语句的具体例子，请参阅附录A中给出的样例表创建脚本。

17.1.1　表创建基础

利用`CREATE TABLE`创建表，必须给出下列信息：

- 新表的名字，在关键字`CREATE TABLE`之后给出；
- 表列的名字和定义，用逗号分隔；

- 有的DBMS还要求指定表的位置。

下面的SQL语句创建本书中所用的Products表：

输入▼

```
CREATE TABLE Products
(
    prod_id        CHAR(10)         NOT NULL,
    vend_id        CHAR(10)         NOT NULL,
    prod_name      CHAR(254)        NOT NULL,
    prod_price     DECIMAL(8,2)     NOT NULL,
    prod_desc      VARCHAR(1000)    NULL
);
```

分析▼
从上面的例子可以看到，表名紧跟CREATE TABLE关键字。实际的表定义（所有列）括在圆括号之中，各列之间用逗号分隔。这个表由5列组成。每列的定义以列名（它在表中必须是唯一的）开始，后跟列的数据类型（关于数据类型的解释，请参阅第1课。此外，附录D列出了常见的数据类型及兼容性）。整条语句以圆括号后的分号结束。

前面提到，不同DBMS的CREATE TABLE的语法有所不同，这个简单脚本也说明了这一点。这条语句在Oracle、PostgreSQL、SQL Server和SQLite中有效，而对于MySQL，varchar必须替换为text；对于DB2，必须从最后一列中去掉NULL。这就是对于不同的DBMS，要编写不同的表创建脚本的原因（参见附录A）。

> 提示：语句格式化
> 回想一下在SQL语句中忽略的空格。语句可以在一个长行上输入，也可以分成许多行，它们没有差别。这样，你就可以用最适合自己的方式安排语句的格式。前面的CREATE TABLE语句就是SQL语句格式化的一个好例子，代码安排在多个行上，列定义进行了恰当的缩进，更易阅读和编辑。以何种格式安排SQL语句并没有规定，但我强烈推荐采用某种缩进格式。

> 提示：替换现有的表
> 在创建新的表时，指定的表名必须不存在，否则会出错。防止意外覆盖已有的表，SQL要求首先手工删除该表（请参阅后面的内

容），然后再重建它，而不是简单地用创建表语句覆盖它。

17.1.2　使用NULL值

第4课提到，NULL值就是没有值或缺值。允许NULL值的列也允许在插入行时不给出该列的值。不允许NULL值的列不接受没有列值的行，换句话说，在插入或更新行时，该列必须有值。

每个表列要么是NULL列，要么是NOT NULL列，这种状态在创建时由表的定义规定。请看下面的例子：

输入▼

```
CREATE TABLE Orders
(
    order_num       INTEGER       NOT NULL,
    order_date      DATETIME      NOT NULL,
    cust_id         CHAR(10)      NOT NULL
);
```

分析▼ 这条语句创建本书中所用的Orders表。Orders包含三列：订单号、订单日期和顾客ID。这三列都需要，因此每一列的定义都含有关键字NOT NULL。这就会阻止插入没有值的列。如果插入没有值的列，将返回错误，且插入失败。

下一个例子将创建混合了NULL和NOT NULL列的表：

输入▼

```
CREATE TABLE Vendors
(
    vend_id          CHAR(10)      NOT NULL,
    vend_name        CHAR(50)      NOT NULL,
    vend_address     CHAR(50)      ,
    vend_city        CHAR(50)      ,
    vend_state       CHAR(5)       ,
    vend_zip         CHAR(10)      ,
    vend_country     CHAR(50)
);
```

分析▼

这条语句创建本书中使用的Vendors表。供应商ID和供应商名字列是必需的，因此指定为NOT NULL。其余五列全都允许NULL值，所以不指定NOT NULL。NULL为默认设置，如果不指定NOT NULL，就认为指定的是NULL。

> 注意：指定**NULL**
> 在不指定NOT NULL时，多数DBMS认为指定的是NULL，但不是所有的DBMS都这样。某些 DBMS 要求指定关键字NULL，如果不指定将出错。关于完整的语法信息，请参阅具体的DBMS文档。

> 提示：主键和**NULL**值
> 第1课介绍过，主键是其值唯一标识表中每一行的列。只有不允许NULL值的列可作为主键，允许NULL值的列不能作为唯一标识。

> 注意：理解**NULL**
> 不要把NULL值与空字符串相混淆。NULL值是没有值，不是空字符串。如果指定''（两个单引号，其间没有字符），这在NOT NULL列中是允许的。空字符串是一个有效的值，它不是无值。NULL值用关键字NULL而不是空字符串指定。

17.1.3　指定默认值

SQL允许指定默认值，在插入行时如果不给出值，DBMS将自动采用默认值。默认值在CREATE TABLE语句的列定义中用关键字DEFAULT指定。

请看下面的例子：

输入▼

```
CREATE TABLE OrderItems
(
    order_num      INTEGER          NOT NULL,
    order_item     INTEGER          NOT NULL,
    prod_id        CHAR(10)         NOT NULL,
```

```
    quantity        INTEGER          NOT NULL      DEFAULT 1,
    item_price      DECIMAL(8,2)     NOT NULL
);
```

分析▼

这条语句创建OrderItems表，包含构成订单的各项（订单本身存储在Orders表中）。quantity列为订单中每个物品的数量。在这个例子中，这一列的描述增加了DEFAULT 1，指示DBMS，如果不给出数量则使用数量1。

默认值经常用于日期或时间戳列。例如，通过指定引用系统日期的函数或变量，将系统日期用作默认日期。MySQL用户指定DEFAULT CURRENT_DATE()，Oracle用户指定DEFAULT SYSDATE，而SQL Server用户指定DEFAULT GETDATE()。遗憾的是，这条获得系统日期的命令在不同的DBMS中几乎都是不同的。表17-1列出了这条命令在某些DBMS中的语法。这里若未列出某个DBMS，请参阅相应的文档。

表**17-1** 获得系统日期

DBMS	函数/变量
Access	NOW()
DB2	CURRENT_DATE
MySQL	CURRENT_DATE()
Oracle	SYSDATE
PostgreSQL	CURRENT_DATE
SQL Server	GETDATE()
SQLite	date('now')

> 提示：使用**DEFAULT**而不是**NULL**值
> 许多数据库开发人员喜欢使用DEFAULT值而不是NULL列，对于用于计算或数据分组的列更是如此。

17.2　更新表

更新表定义，可以使用ALTER TABLE语句。虽然所有的DBMS都支持ALTER TABLE，但它们所允许更新的内容差别很大。以下是使用ALTER TABLE时需要考虑的事情。

- 理想情况下，不要在表中包含数据时对其进行更新。应该在表

的设计过程中充分考虑未来可能的需求，避免今后对表的结构做大改动。
- 所有的DBMS都允许给现有的表增加列，不过对所增加列的数据类型（以及NULL和DEFAULT的使用）有所限制。
- 许多DBMS不允许删除或更改表中的列。
- 多数DBMS允许重新命名表中的列。
- 许多DBMS限制对已经填有数据的列进行更改，对未填有数据的列几乎没有限制。

可以看出，对已有表做更改既复杂又不统一。对表的结构能进行何种更改，请参阅具体的DBMS文档。

使用ALTER TABLE更改表结构，必须给出下面的信息：

- 在ALTER TABLE之后给出要更改的表名（该表必须存在，否则将出错）；
- 列出要做哪些更改。

因为给已有表增加列可能是所有DBMS都支持的唯一操作，所以我们举个这样的例子：

输入▼

```
ALTER TABLE Vendors
ADD vend_phone CHAR(20);
```

分析▼
这条语句给Vendors表增加一个名为vend_phone的列，其数据类型为CHAR。

更改或删除列、增加约束或增加键，这些操作也使用类似的语法（注意，下面的例子并非对所有DBMS都有效）：

输入▼

```
ALTER TABLE Vendors
DROP COLUMN vend_phone;
```

复杂的表结构更改一般需要手动删除过程，它涉及以下步骤：

1. 用新的列布局创建一个新表；

2. 使用INSERT SELECT语句（关于这条语句的详细介绍，请参阅第15课）从旧表复制数据到新表。有必要的话，可以使用转换函数和计算字段；
3. 检验包含所需数据的新表；
4. 重命名旧表（如果确定，可以删除它）；
5. 用旧表原来的名字重命名新表；
6. 根据需要，重新创建触发器、存储过程、索引和外键。

> 说明：**ALTER TABLE和SQLite**
> SQLite对使用ALTER TABLE执行的操作有所限制。最重要的一个限制是，它不支持使用ALTER TABLE定义主键和外键，这些必须在最初创建表时指定。

> 注意：小心使用**ALTER TABLE**
> 使用ALTER TABLE要极为小心，应该在进行改动前做完整的备份（表结构和数据的备份）。数据库表的更改不能撤销，如果增加了不需要的列，也许无法删除它们。类似地，如果删除了不应该删除的列，可能会丢失该列中的所有数据。

17.3 删除表

删除表（删除整个表而不是其内容）非常简单，使用DROP TABLE语句即可：

输入▼

```
DROP TABLE CustCopy;
```

分析▼
这条语句删除CustCopy表（第15课中创建的）。删除表没有确认，也不能撤销，执行这条语句将永久删除该表。

> 提示：使用关系规则防止意外删除
> 许多DBMS允许强制实施有关规则，防止删除与其他表相关联的表。在实施这些规则时，如果对某个表发布一条DROP TABLE语句，且该表是某个关系的组成部分，则DBMS将阻止这条语句执行，直到该关系被删除为止。如果允许，应该启用这些选项，它

能防止意外删除有用的表。

17.4　重命名表

每个DBMS对表重命名的支持有所不同。对于这个操作，不存在严格的标准。DB2、MariaDB、MySQL、Oracle和PostgreSQL用户使用`RENAME`语句，SQL Server用户使用`sp_rename`存储过程，SQLite用户使用`ALTER TABLE`语句。

所有重命名操作的基本语法都要求指定旧表名和新表名。不过，存在DBMS实现差异。关于具体的语法，请参阅相应的DBMS文档。

17.5　小结

这一课介绍了几条新的SQL语句。`CREATE TABLE`用来创建新表，`ALTER TABLE`用来更改表列（或其他诸如约束或索引等对象），而`DROP TABLE`用来完整地删除一个表。这些语句必须小心使用，并且应该在备份后使用。由于这些语句的语法在不同的DBMS中有所不同，所以更详细的信息请参阅相应的DBMS文档。

第18课　使用视图

这一课将介绍什么是视图，它们怎样工作，何时使用它们；还将讲述如何利用视图简化前几课中执行的某些SQL操作。

18.1　视图

视图是虚拟的表。与包含数据的表不一样，视图只包含使用时动态检索数据的查询。

> 说明：**DBMS**支持
> Microsoft Access不支持视图，没有与SQL视图一致的工作方式。因此，这一课的内容不适用Microsoft Access。
>
> MySQL从版本5起开始支持视图，因此，这一课的内容不适用较早版本的MySQL。
>
> SQLite仅支持只读视图，所以视图可以创建，可以读，但其内容不能更改。

理解视图的最好方法是看例子。第12课用下面的SELECT语句从三个表中检索数据：

输入▼

```
SELECT cust_name, cust_contact
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
 AND OrderItems.order_num = Orders.order_num
 AND prod_id = 'RGAN01';
```

此查询用来检索订购了某种产品的顾客。任何需要这个数据的人都必须理解相关表的结构，知道如何创建查询和对表进行联结。检索其他产品（或多个产品）的相同数据，必须修改最后的WHERE子句。

现在，假如可以把整个查询包装成一个名为ProductCustomers的虚拟表，则可以如下轻松地检索出相同的数据：

输入▼

```
SELECT cust_name, cust_contact
FROM ProductCustomers
WHERE prod_id = 'RGAN01';
```

这就是视图的作用。ProductCustomers是一个视图，作为视图，它不包含任何列或数据，包含的是一个查询（与上面用以正确联结表的查询相同）。

> 提示：**DBMS**的一致支持
> 我们欣慰地了解到，所有DBMS非常一致地支持视图创建语法。

18.1.1　为什么使用视图

我们已经看到了视图应用的一个例子。下面是视图的一些常见应用。

- 重用SQL语句。
- 简化复杂的SQL操作。在编写查询后，可以方便地重用它而不必知道其基本查询细节。
- 使用表的一部分而不是整个表。
- 保护数据。可以授予用户访问表的特定部分的权限，而不是整个表的访问权限。
- 更改数据格式和表示。视图可返回与底层表的表示和格式不同的数据。

创建视图之后，可以用与表基本相同的方式使用它们。可以对视图执行SELECT操作，过滤和排序数据，将视图联结到其他视图或表，甚至添加和更新数据（添加和更新数据存在某些限制，关于这个内容稍后做介绍）。

重要的是，要知道视图仅仅是用来查看存储在别处数据的一种设施。视图本身不包含数据，因此返回的数据是从其他表中检索出来的。在添加或更改这些表中的数据时，视图将返回改变过的数据。

> 警告：性能问题
> 因为视图不包含数据，所以每次使用视图时，都必须处理查询执行时需要的所有检索。如果你用多个联结和过滤创建了复杂的视图或者嵌套了视图，性能可能会下降得很厉害。因此，在部署使用了大量视图的应用前，应该进行测试。

18.1.2　视图的规则和限制

创建视图前，应该知道它的一些限制。不过，这些限制随不同的DBMS而不同，因此在创建视图时应该查看具体的DBMS文档。

下面是关于视图创建和使用的一些最常见的规则和限制。

- 与表一样，视图必须唯一命名（不能给视图取与别的视图或表相同的名字）。
- 对于可以创建的视图数目没有限制。
- 创建视图，必须具有足够的访问权限。这些权限通常由数据库管理人员授予。
- 视图可以嵌套，即可以利用从其他视图中检索数据的查询来构造视图。所允许的嵌套层数在不同的DBMS中有所不同（嵌套视图可能会严重降低查询的性能，因此在产品环境中使用之前，应该对其进行全面测试）。
- 许多DBMS禁止在视图查询中使用`ORDER BY`子句。
- 有些DBMS要求对返回的所有列进行命名，如果列是计算字段，则需要使用别名（关于列别名的更多信息，请参阅第7课）。
- 视图不能索引，也不能有关联的触发器或默认值。
- 有些DBMS把视图作为只读的查询，这表示可以从视图检索数据，但不能将数据写回底层表。详情请参阅具体的DBMS文档。
- 有些DBMS允许创建这样的视图，它不能进行导致行不再属于视图的插入或更新。例如有一个视图，只检索带有电子邮件地址的顾客。如果更新某个顾客，删除他的电子邮件地址，将使该顾客不再属于视图。这是默认行为，而且是允许的，但有的DBMS可能会防止这种情况发生。

> 提示：参阅具体的**DBMS**文档
> 上面的规则不少，而具体的DBMS文档很可能还包含别的规则。因此，在创建视图前，有必要花点时间了解必须遵守的规定。

18.2　创建视图

理解了什么是视图以及管理它们的规则和约束后，我们来创建视图。

视图用`CREATE VIEW`语句来创建。与`CREATE TABLE`一样，`CREATE VIEW`只能用于创建不存在的视图。

> 说明：视图重命名
> 删除视图，可以使用DROP语句，其语法为DROP VIEW viewname;。
>
> 覆盖（或更新）视图，必须先删除它，然后再重新创建。

18.2.1 利用视图简化复杂的联结

一个最常见的视图应用是隐藏复杂的SQL，这通常涉及联结。请看下面的例子：

输入▼

```
CREATE VIEW ProductCustomers AS
SELECT cust_name, cust_contact, prod_id
FROM Customers, Orders, OrderItems
WHERE Customers.cust_id = Orders.cust_id
 AND OrderItems.order_num = Orders.order_num;
```

分析▼
这条语句创建一个名为ProductCustomers的视图，它联结三个表，返回已订购了任意产品的所有顾客的列表。如果执行SELECT * FROM ProductCustomers，将列出订购了任意产品的顾客。

检索订购了产品RGAN01的顾客，可如下进行：

输入▼

```
SELECT cust_name, cust_contact
FROM ProductCustomers
WHERE prod_id = 'RGAN01';
```

输出▼

```
cust_name                  cust_contact
-------------------        ------------------
Fun4All                    Denise L. Stephens
The Toy Store              Kim Howard
```

分析▼
这条语句通过WHERE子句从视图中检索特定数据。当DBMS处理此查

询时，它将指定的WHERE子句添加到视图查询中已有的WHERE子句中，以便正确过滤数据。

可以看出，视图极大地简化了复杂SQL语句的使用。利用视图，可一次性编写基础的SQL，然后根据需要多次使用。

> 提示：创建可重用的视图
> 创建不绑定特定数据的视图是一种好办法。例如，上面创建的视图返回订购所有产品而不仅仅是RGAN01的顾客（这个视图先创建）。扩展视图的范围不仅使得它能被重用，而且可能更有用。这样做不需要创建和维护多个类似视图。

18.2.2 用视图重新格式化检索出的数据

如前所述，视图的另一常见用途是重新格式化检索出的数据。下面的SELECT语句（来自第7课）在单个组合计算列中返回供应商名和位置：

输入▼

```
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

输出▼

```
vend_title
-----------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

下面是相同的语句，但使用了||语法（如第7课所述）：

输入▼

```
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
```

```
       AS vend_title
FROM Vendors
ORDER BY vend_name;
```

输出▼

```
vend_title
-----------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

现在，假设经常需要这个格式的结果。我们不必在每次需要时执行这种拼接，而是创建一个视图，使用它即可。把此语句转换为视图，可按如下进行：

输入▼

```
CREATE VIEW VendorLocations AS
SELECT RTRIM(vend_name) + ' (' + RTRIM(vend_country) + ')'
       AS vend_title
FROM Vendors;
```

下面是使用||语法的相同语句：

输入▼

```
CREATE VIEW VendorLocations AS
SELECT RTRIM(vend_name) || ' (' || RTRIM(vend_country) || ')'
AS vend_title
FROM Vendors;
```

分析▼

这条语句使用与以前SELECT语句相同的查询创建视图。要检索数据，创建所有的邮件标签，可如下进行：

输入▼

```
SELECT *
```

```
FROM VendorLocations;
```

输出▼

```
vend_title
-----------------------------------------------------------
Bear Emporium (USA)
Bears R Us (USA)
Doll House Inc. (USA)
Fun and Games (England)
Furball Inc. (USA)
Jouets et ours (France)
```

> 说明：**SELECT**约束全部适用
> 在这一课的前面提到，各种DBMS中用来创建视图的语法相当一致。那么，为什么会有多种创建视图的语句版本呢？因为视图只包含一个SELECT语句，而这个语句的语法必须遵循具体DBMS的所有规则和约束，所以会有多个创建视图的语句版本。

18.2.3　用视图过滤不想要的数据

视图对于应用普通的WHERE子句也很有用。例如，可以定义CustomerEMailList视图，过滤没有电子邮件地址的顾客。为此，可使用下面的语句：

输入▼

```
CREATE VIEW CustomerEMailList AS
SELECT cust_id, cust_name, cust_email
FROM Customers
WHERE cust_email IS NOT NULL;
```

分析▼
显然，在将电子邮件发送到邮件列表时，需要排除没有电子邮件地址的用户。这里的WHERE子句过滤了cust_email列中具有NULL值的那些行，使它们不被检索出来。

现在，可以像使用其他表一样使用视图CustomerEMailList。

输入▼

```
```

```
SELECT *
FROM CustomerEMailList;
```

输出▼

```
cust_id         cust_name        cust_email
----------      ------------     ---------------------
1000000001      Village Toys     sales@villagetoys.com
1000000003      Fun4All          jjones@fun4all.com
1000000004      Fun4All          dstephens@fun4all.com
```

> 说明：**WHERE**子句与**WHERE**子句
> 从视图检索数据时如果使用了一条WHERE子句，则两组子句（一组在视图中，另一组是传递给视图的）将自动组合。

18.2.4　使用视图与计算字段

在简化计算字段的使用上，视图也特别有用。下面是第7课中介绍的一条SELECT语句，它检索某个订单中的物品，计算每种物品的总价格：

输入▼

```
SELECT prod_id,
       quantity,
       item_price,
       quantity*item_price AS expanded_price
FROM OrderItems
WHERE order_num = 20008;
```

输出▼

```
prod_id       quantity      item_price     expanded_price
--------      ---------     -----------    ---------------
RGAN01        5             4.9900         24.9500
BR03          5             11.9900        59.9500
BNBG01        10            3.4900         34.9000
BNBG02        10            3.4900         34.9000
BNBG03        10            3.4900         34.9000
```

要将其转换为一个视图，如下进行：

输入▼

```
CREATE VIEW OrderItemsExpanded AS
SELECT order_num,
       prod_id,
       quantity,
       item_price,
       quantity*item_price AS expanded_price
FROM OrderItems;
```

检索订单20008的详细内容（上面的输出），如下进行：

输入▼

```
SELECT *
FROM OrderItemsExpanded
WHERE order_num = 20008;
```

输出▼

```
order_num     prod_id     quantity     item_price     expand
---------     -------     ---------    ----------     ------
20008         RGAN01      5            4.99           24.95
20008         BR03        5            11.99          59.95
20008         BNBG01      10           3.49           34.90
20008         BNBG02      10           3.49           34.90
20008         BNBG03      10           3.49           34.90
```

可以看到，视图非常容易创建，而且很好使用。正确使用，视图可极大地简化复杂数据的处理。

18.3　小结

视图为虚拟的表。它们包含的不是数据而是根据需要检索数据的查询。视图提供了一种封装SELECT语句的层次，可用来简化数据处理，重新格式化或保护基础数据。

第19课　使用存储过程

这一课介绍什么是存储过程，为什么要使用存储过程，如何使用存储过程，以及创建和使用存储过程的基本语法。

19.1　存储过程

迄今为止，我们使用的大多数SQL语句都是针对一个或多个表的单条语句。并非所有操作都这么简单，经常会有一些复杂的操作需要多条语句才能完成。例如以下的情形：

- 为了处理订单，必须核对以保证库存中有相应的物品。
- 如果物品有库存，需要预定，不再出售给别的人，并且减少物品数据以反映正确的库存量。
- 库存中没有的物品需要订购，这需要与供应商进行某种交互。
- 关于哪些物品入库（并且可以立即发货）和哪些物品退订，需要通知相应的顾客。

这显然不是一个完整的例子，它甚至超出了本书中所用样例表的范围，但足以表达我们的意思了。执行这个处理需要针对许多表的多条SQL语句。此外，需要执行的具体SQL语句及其次序也不是固定的，它们可能会根据物品是否在库存中而变化。

那么，怎样编写代码呢？可以单独编写每条SQL语句，并根据结果有条件地执行其他语句。在每次需要这个处理时（以及每个需要它的应用中），都必须做这些工作。

可以创建存储过程。简单来说，存储过程就是为以后使用而保存的一条或多条SQL语句。可将其视为批文件，虽然它们的作用不仅限于批处理。

> 说明：具体**DBMS**的支持
> Microsoft Access和SQLite不支持存储过程。因此，本课的内容不适用它们。
>
> MySQL 5已经支持存储过程。因此，本课的内容不适用MySQL较早的版本。

> 说明：还有更多内容
> 存储过程很复杂，全面介绍它需要很大篇幅。本课不打算讲解存储过程的所有内容，只给出简单介绍，让读者对它们的功能有所了解。因此，这里给出的例子只提供Oracle和SQL Server的语法。

19.2　为什么要使用存储过程

我们知道了什么是存储过程，那么为什么要使用它们呢？理由很多，下面列出一些主要的。

- 通过把处理封装在一个易用的单元中，可以简化复杂的操作（如前面例子所述）。
- 由于不要求反复建立一系列处理步骤，因而保证了数据的一致性。如果所有开发人员和应用程序都使用同一存储过程，则所使用的代码都是相同的。
  这一点的延伸就是防止错误。需要执行的步骤越多，出错的可能性就越大。防止错误保证了数据的一致性。
- 简化对变动的管理。如果表名、列名或业务逻辑（或别的内容）有变化，那么只需要更改存储过程的代码。使用它的人员甚至不需要知道这些变化。
  这一点的延伸就是安全性。通过存储过程限制对基础数据的访问，减少了数据讹误（无意识的或别的原因所导致的数据讹误）的机会。
- 因为存储过程通常以编译过的形式存储，所以DBMS处理命令所需的工作量较少，提高了性能。
- 存在一些只能用在单个请求中的SQL元素和特性，存储过程可以使用它们来编写功能更强更灵活的代码。

换句话说，使用存储过程有三个主要的好处，即简单、安全、高性能。显然，它们都很重要。不过，在将SQL代码转换为存储过程前，也必须知道它的一些缺陷。

- 不同DBMS中的存储过程语法有所不同。事实上，编写真正的

可移植存储过程几乎是不可能的。不过，存储过程的自我调用（名字以及数据如何传递）可以相对保持可移植。因此，如果需要移植到别的DBMS，至少客户端应用代码不需要变动。

- 一般来说，编写存储过程比编写基本SQL语句复杂，需要更高的技能，更丰富的经验。因此，许多数据库管理员把限制存储过程的创建作为安全措施（主要受上一条缺陷的影响）。

尽管有这些缺陷，存储过程还是非常有用的，并且应该使用。事实上，多数DBMS都带有用于管理数据库和表的各种存储过程。更多信息请参阅具体的DBMS文档。

> 说明：不能编写存储过程？你依然可以使用
> 大多数DBMS将编写存储过程所需的安全和访问权限与执行存储过程所需的安全和访问权限区分开来。这是好事情，即使你不能（或不想）编写自己的存储过程，也仍然可以在适当的时候执行别的存储过程。

19.3　执行存储过程

存储过程的执行远比编写要频繁得多，因此我们先介绍存储过程的执行。执行存储过程的SQL语句很简单，即EXECUTE。EXECUTE接受存储过程名和需要传递给它的任何参数。请看下面的例子：

输入▼

```
EXECUTE AddNewProduct( 'JTS01',
                       'Stuffed Eiffel Tower',
                       6.49,
                       'Plush stuffed toy with the text La
➥Tour Eiffel in red white and blue' );
```

分析▼
这里执行一个名为AddNewProduct的存储过程，将一个新产品添加到Products表中。AddNewProduct有四个参数，分别是：供应商ID（Vendors表的主键）、产品名、价格和描述。这4个参数匹配存储过程中4个预期变量（定义为存储过程自身的组成部分）。此存储过程将新行添加到Products表，并将传入的属性赋给相应的列。

我们注意到，在Products表中还有另一个需要值的列prod_id列，

它是这个表的主键。为什么这个值不作为属性传递给存储过程？要保证恰当地生成此ID，最好是使生成此ID的过程自动化（而不是依赖于最终用户的输入）。这也是这个例子使用存储过程的原因。以下是存储过程所完成的工作：

- 验证传递的数据，保证所有4个参数都有值；
- 生成用作主键的唯一ID；
- 将新产品插入`Products`表，在合适的列中存储生成的主键和传递的数据。

这就是存储过程执行的基本形式。对于具体的DBMS，可能包括以下的执行选择：

- 参数可选，具有不提供参数时的默认值；
- 不按次序给出参数，以“参数＝值”的方式给出参数值。
- 输出参数，允许存储过程在正执行的应用程序中更新所用的参数。
- 用`SELECT`语句检索数据。
- 返回代码，允许存储过程返回一个值到正在执行的应用程序。

19.4　创建存储过程

正如所述，存储过程的编写很重要。为了获得感性认识，我们来看一个简单的存储过程例子，它对邮件发送清单中具有邮件地址的顾客进行计数。

下面是该过程的Oracle版本：

输入▼

```
CREATE PROCEDURE MailingListCount (
  ListCount OUT INTEGER
)
IS
v_rows INTEGER;
BEGIN
    SELECT COUNT(*) INTO v_rows
    FROM Customers
    WHERE NOT cust_email IS NULL;
    ListCount := v_rows;
END;
```

分析▼
这个存储过程有一个名为ListCount的参数。此参数从存储过程返回一个值而不是传递一个值给存储过程。关键字OUT用来指示这种行为。Oracle提供IN（传递值给存储过程）、OUT（从存储过程返回值，如这里）、INOUT（既传递值给存储过程也从存储过程传回值）类型的参数。存储过程的代码括在BEGIN和END语句中，这里执行一条简单的SELECT语句，它检索具有邮件地址的顾客。然后用检索出的行数设置ListCount（要传递的输出参数）。

调用Oracle例子可以像下面这样：

输入▼

```
var ReturnValue NUMBER
EXEC MailingListCount(:ReturnValue);
SELECT ReturnValue;
```

分析▼
这段代码声明了一个变量来保存存储过程返回的任何值，然后执行存储过程，再使用SELECT语句显示返回的值。

下面是该过程的SQL Server版本：

输入▼

```
CREATE PROCEDURE MailingListCount
AS
DECLARE @cnt INTEGER
SELECT @cnt = COUNT(*)
FROM Customers
WHERE NOT cust_email IS NULL;
RETURN @cnt;
```

分析▼
此存储过程没有参数。调用程序检索SQL Server的返回代码提供的值。其中用DECLARE语句声明了一个名为@cnt的局部变量（SQL Server中所有局部变量名都以@起头）；然后在SELECT语句中使用这个变量，让它包含COUNT()函数返回的值；最后，用RETURN @cnt语句将计数返回给调用程序。

调用SQL Server例子可以像下面这样：

输入▼

```
DECLARE @ReturnValue INT
EXECUTE @ReturnValue=MailingListCount;
SELECT @ReturnValue;
```

分析▼
这段代码声明了一个变量来保存存储过程返回的任何值，然后执行存储过程，再使用SELECT语句显示返回的值。

下面是另一个例子，这次在Orders表中插入一个新订单。此程序仅适用于SQL Server，但它说明了存储过程的某些用途和技术：

输入▼

```
CREATE PROCEDURE NewOrder @cust_id CHAR(10)
AS
-- Declare variable for order number
DECLARE @order_num INTEGER
-- Get current highest order number
SELECT @order_num=MAX(order_num)
FROM Orders
-- Determine next order number
SELECT @order_num=@order_num+1
-- Insert new order
INSERT INTO Orders(order_num, order_date, cust_id)
VALUES(@order_num, GETDATE(), @cust_id)
-- Return order number
RETURN @order_num;
```

分析▼
此存储过程在Orders表中创建一个新订单。它只有一个参数，即下订单顾客的ID。订单号和订单日期这两列在存储过程中自动生成。代码首先声明一个局部变量来存储订单号。接着，检索当前最大订单号（使用MAX()函数）并增加1（使用SELECT语句）。然后用INSERT语句插入由新生成的订单号、当前系统日期（用GETDATE()函数检索）和传递的顾客ID组成的订单。最后，用RETURN @order_num返回订单号（处理订单物品需要它）。请注意，此代码加了注释，在编写存储过程时应该多加注释。

> 说明：注释代码
> 应该注释所有代码，存储过程也不例外。增加注释不影响性能，

> 因此不存在缺陷（除了增加编写时间外）。注释代码的好处很多，包括使别人（以及你自己）更容易地理解和更安全地修改代码。
>
> 对代码进行注释的标准方式是在之前放置--（两个连字符）。有的DBMS还支持其他的注释语法，不过所有DBMS都支持--，因此在注释代码时最好都使用这种语法。

下面是相同SQL Server代码的一个很不同的版本：

输入▼

```
CREATE PROCEDURE NewOrder @cust_id CHAR(10)
AS
-- Insert new order
INSERT INTO Orders(cust_id)
VALUES(@cust_id)
-- Return order number
SELECT order_num = @@IDENTITY;
```

分析▼
此存储过程也在`Orders`表中创建一个新订单。这次由DBMS生成订单号。大多数DBMS都支持这种功能；SQL Server中称这些自动增量的列为标识字段（identity field），而其他DBMS称之为自动编号（auto number）或序列（sequence）。传递给此过程的参数也是一个，即下订单的顾客ID。订单号和订单日期没有给出，DBMS对日期使用默认值（`GETDATE()`函数），订单号自动生成。怎样才能得到这个自动生成的ID？在SQL Server上可在全局变量`@@IDENTITY`中得到，它返回到调用程序（这里使用`SELECT`语句）。

可以看到，借助存储过程，可以有多种方法完成相同的工作。不过，所选择的方法受所用DBMS特性的制约。

19.5　小结

这一课介绍了什么是存储过程，为什么使用存储过程。我们介绍了执行和创建存储过程的语法，使用存储过程的一些方法。存储过程是个相当重要的主题，一课内容无法全部涉及。各种DBMS对存储过程的实现不一，你使用的DBMS可能提供了一些这里提到的功能，也有其他未提及的功能，更详细的介绍请参阅具体的DBMS文档。

第20课　管理事务处理

这一课介绍什么是事务处理，如何利用COMMIT和ROLLBACK语句管理事务处理。

20.1　事务处理

使用事务处理（transaction processing），通过确保成批的SQL操作要么完全执行，要么完全不执行，来维护数据库的完整性。

正如第12课所述，关系数据库把数据存储在多个表中，使数据更容易操纵、维护和重用。不用深究如何以及为什么进行关系数据库设计，在某种程度上说，设计良好的数据库模式都是关联的。

前面使用的Orders表就是一个很好的例子。订单存储在Orders和OrderItems两个表中：Orders存储实际的订单，OrderItems存储订购的各项物品。这两个表使用称为主键（参阅第1课）的唯一ID互相关联，又与包含客户和产品信息的其他表相关联。

给系统添加订单的过程如下：

1. 检查数据库中是否存在相应的顾客，如果不存在，添加他；
2. 检索顾客的ID；
3. 在Orders表添加一行，它与顾客ID相关联；
4. 检索Orders表中赋予的新订单ID；
5. 为订购的每个物品在OrderItems表中添加一行，通过检索出来的ID把它与Orders表关联（并且通过产品ID与Products表关联）。

现在假设由于某种数据库故障（如超出磁盘空间、安全限制、表锁等），这个过程无法完成。数据库中的数据会出现什么情况？

如果故障发生在添加顾客之后，添加Orders表之前，则不会有什么问题。某些顾客没有订单是完全合法的。重新执行此过程时，所插入的顾客记录将被检索和使用。可以有效地从出故障的地方开始执行此过程。

但是，如果故障发生在插入Orders行之后，添加OrderItems行之

前，怎么办？现在，数据库中有一个空订单。

更糟的是，如果系统在添加`OrderItems`行之时出现故障，怎么办？结果是数据库中存在不完整的订单，而你还不知道。

如何解决这种问题？这就需要使用事务处理了。事务处理是一种机制，用来管理必须成批执行的SQL操作，保证数据库不包含不完整的操作结果。利用事务处理，可以保证一组操作不会中途停止，它们要么完全执行，要么完全不执行（除非明确指示）。如果没有错误发生，整组语句提交给（写到）数据库表；如果发生错误，则进行回退（撤销），将数据库恢复到某个已知且安全的状态。

再看这个例子，这次我们说明这一过程是如何工作的：

1. 检查数据库中是否存在相应的顾客，如果不存在，添加他；
2. 提交顾客信息；
3. 检索顾客的ID；
4. 在`Orders`表中添加一行；
5. 如果向`Orders`表添加行时出现故障，回退；
6. 检索`Orders`表中赋予的新订单ID；
7. 对于订购的每项物品，添加新行到`OrderItems`表；
8. 如果向`OrderItems`添加行时出现故障，回退所有添加的`OrderItems`行和`Orders`行。

在使用事务处理时，有几个反复出现的关键词。下面是关于事务处理需要知道的几个术语：

- 事务（transaction）指一组SQL语句；
- 回退（rollback）指撤销指定SQL语句的过程；
- 提交（commit）指将未存储的SQL语句结果写入数据库表；
- 保留点（savepoint）指事务处理中设置的临时占位符（placeholder），可以对它发布回退（与回退整个事务处理不同）。

> 提示：可以回退哪些语句？
> 事务处理用来管理`INSERT`、`UPDATE`和`DELETE`语句。不能回退`SELECT`语句（回退`SELECT`语句也没有必要），也不能回退`CREATE`或`DROP`操作。事务处理中可以使用这些语句，但进行回退时，这些操作也不撤销。

20.2 控制事务处理

我们已经知道了什么是事务处理，下面讨论管理事务中涉及的问题。

> 警告：事务处理实现的差异
> 不同DBMS用来实现事务处理的语法有所不同。在使用事务处理时请参阅相应的DBMS文档。

管理事务的关键在于将SQL语句组分解为逻辑块，并明确规定数据何时应该回退，何时不应该回退。

有的DBMS要求明确标识事务处理块的开始和结束。如在SQL Server中，标识如下：

输入▼

```
BEGIN TRANSACTION
...
COMMIT TRANSACTION
```

分析▼
在这个例子中，`BEGIN TRANSACTION`和`COMMIT TRANSACTION`语句之间的SQL必须完全执行或者完全不执行。

MariaDB和MySQL中等同的代码为：

输入▼

```
START TRANSACTION
...
```

Oracle使用的语法：

输入▼

```
SET TRANSACTION
...
```

PostgreSQL使用ANSI SQL语法：

输入▼

```
BEGIN
...
```

其他DBMS采用上述语法的变体。你会发现，多数实现没有明确标识事务处理在何处结束。事务一直存在，直到被中断。通常，COMMIT用于保存更改，ROLLBACK用于撤销，详述如下。

20.2.1 使用ROLLBACK

SQL的ROLLBACK命令用来回退（撤销）SQL语句，请看下面的语句：

输入▼

```
DELETE FROM Orders;
ROLLBACK;
```

分析▼
在此例子中，执行DELETE操作，然后用ROLLBACK语句撤销。虽然这不是最有用的例子，但它的确能够说明，在事务处理块中，DELETE操作（与INSERT和UPDATE操作一样）并不是最终的结果。

20.2.2 使用COMMIT

一般的SQL语句都是针对数据库表直接执行和编写的。这就是所谓的隐式提交（implicit commit），即提交（写或保存）操作是自动进行的。

在事务处理块中，提交不会隐式进行。不过，不同DBMS的做法有所不同。有的DBMS按隐式提交处理事务端，有的则不这样。

进行明确的提交，使用COMMIT语句。下面是一个SQL Server的例子：

输入▼

```
BEGIN TRANSACTION
DELETE OrderItems WHERE order_num = 12345
DELETE Orders WHERE order_num = 12345

```

```
COMMIT TRANSACTION
```

分析▼

在这个SQL Server例子中，从系统中完全删除订单12345。因为涉及更新两个数据库表Orders和OrderItems，所以使用事务处理块来保证订单不被部分删除。最后的COMMIT语句仅在不出错时写出更改。如果第一条DELETE起作用，但第二条失败，则DELETE不会提交。

为在Oracle中完成相同的工作，可如下进行：

输入▼

```
SET TRANSACTION
DELETE OrderItems WHERE order_num = 12345;
DELETE Orders WHERE order_num = 12345;
COMMIT;
```

20.2.3　使用保留点

使用简单的ROLLBACK和COMMIT语句，就可以写入或撤销整个事务。但是，只对简单的事务才能这样做，复杂的事务可能需要部分提交或回退。

例如前面描述的添加订单的过程就是一个事务。如果发生错误，只需要返回到添加Orders行之前即可。不需要回退到Customers表（如果存在的话）。

要支持回退部分事务，必须在事务处理块中的合适位置放置占位符。这样，如果需要回退，可以回退到某个占位符。

在SQL中，这些占位符称为保留点。在MariaDB、MySQL和Oracle中创建占位符，可使用SAVEPOINT语句：

输入▼

```
SAVEPOINT delete1;
```

在SQL Server中，如下进行：

输入▼

```
SAVE TRANSACTION delete1;
```

每个保留点都要取能够标识它的唯一名字，以便在回退时，DBMS知道回退到何处。要回退到本例给出的保留点，在SQL Server中可如下进行：

输入▼

```
ROLLBACK TRANSACTION delete1;
```

在MariaDB、MySQL和Oracle中，如下进行：

输入▼

```
ROLLBACK TO delete1;
```

下面是一个完整的SQL Server例子：

输入▼

```
BEGIN TRANSACTION
INSERT INTO Customers(cust_id, cust_name)
VALUES('1000000010', 'Toys Emporium');
SAVE TRANSACTION StartOrder;
INSERT INTO Orders(order_num, order_date, cust_id)
VALUES(20100,'2001/12/1','1000000010');
IF @@ERROR <> 0 ROLLBACK TRANSACTION StartOrder;
INSERT INTO OrderItems(order_num, order_item, prod_id, quanti
VALUES(20100, 1, 'BR01', 100, 5.49);
IF @@ERROR <> 0 ROLLBACK TRANSACTION StartOrder;
INSERT INTO OrderItems(order_num, order_item, prod_id, quanti
VALUES(20100, 2, 'BR03', 100, 10.99);
IF @@ERROR <> 0 ROLLBACK TRANSACTION StartOrder;
COMMIT TRANSACTION
```

分析▼

这里的事务处理块中包含了4条`INSERT`语句。在第一条`INSERT`语句之后定义了一个保留点，因此，如果后面的任何一个`INSERT`操作失败，事务处理能够回退到这里。在SQL Server中，可检查一个名为`@@ERROR`的变量，看操作是否成功。（其他DBMS使用不同的函数或变量返回此信息。）如果`@@ERROR`返回一个非`0`的值，表示有错误发

生，事务处理回退到保留点。如果整个事务处理成功，发布COMMIT以保留数据。

> 提示：保留点越多越好
> 可以在SQL代码中设置任意多的保留点，越多越好。为什么呢？因为保留点越多，你就越能灵活地进行回退。

20.3　小结

这一课介绍了事务是必须完整执行的SQL语句块。我们学习了如何使用COMMIT和ROLLBACK语句对何时写数据、何时撤销进行明确的管理；还学习了如何使用保留点，更好地控制回退操作。事务处理是个相当重要的主题，一课内容无法全部涉及。各种DBMS对事务处理的实现不同，详细内容请参考具体的DBMS文档。

第21课　使用游标

这一课将讲授什么是游标，如何使用游标。

21.1　游标

SQL检索操作返回一组称为结果集的行，这组返回的行都是与SQL语句相匹配的行（零行或多行）。简单地使用`SELECT`语句，没有办法得到第一行、下一行或前10行。但这是关系DBMS功能的组成部分。

> 结果集（**result set**）
> SQL查询所检索出的结果。

有时，需要在检索出来的行中前进或后退一行或多行，这就是游标的用途所在。游标（cursor）是一个存储在DBMS服务器上的数据库查询，它不是一条`SELECT`语句，而是被该语句检索出来的结果集。在存储了游标之后，应用程序可以根据需要滚动或浏览其中的数据。

> 说明：具体**DBMS**的支持
> Microsoft Access不支持游标，所以本课的内容不适用于Microsoft Access。
>
> MySQL 5已经支持游标。因此，本课的内容不适用MySQL较早的版本。
>
> SQLite支持的游标称为步骤（step），本课讲述的基本概念适用于SQLite的步骤，但语法可能完全不同。

不同的DBMS支持不同的游标选项和特性。常见的一些选项和特性如下。

- 能够标记游标为只读，使数据能读取，但不能更新和删除。
- 能控制可以执行的定向操作（向前、向后、第一、最后、绝对位置、相对位置等）。
- 能标记某些列为可编辑的，某些列为不可编辑的。
- 规定范围，使游标对创建它的特定请求（如存储过程）或对所有请求可访问。

- 指示DBMS对检索出的数据（而不是指出表中活动数据）进行复制，使数据在游标打开和访问期间不变化。

游标主要用于交互式应用，其中用户需要滚动屏幕上的数据，并对数据进行浏览或做出更改。

> 说明：游标与基于**Web**的应用
> 游标对基于Web的应用（如ASP、ASP.NET、ColdFusion、PHP、Python、Ruby、JSP等）用处不大。虽然游标在客户端应用和服务器会话期间存在，但这种客户/服务器模式不适合Web应用，因为应用服务器是数据库客户端而不是最终用户。所以，大多数Web应用开发人员不使用游标，他们根据自己的需要重新开发相应的功能。

21.2　使用游标

使用游标涉及几个明确的步骤：

- 在使用游标前，必须声明（定义）它。这个过程实际上没有检索数据，它只是定义要使用的`SELECT`语句和游标选项。
- 一旦声明，就必须打开游标以供使用。这个过程用前面定义的`SELECT`语句把数据实际检索出来。
- 对于填有数据的游标，根据需要取出（检索）各行。
- 在结束游标使用时，必须关闭游标，可能的话，释放游标（有赖于具体的DBMS）。

声明游标后，可根据需要频繁地打开和关闭游标。在游标打开时，可根据需要频繁地执行取操作。

21.2.1　创建游标

使用`DECLARE`语句创建游标，这条语句在不同的DBMS中有所不同。`DECLARE`命名游标，并定义相应的`SELECT`语句，根据需要带`WHERE`和其他子句。为了说明，我们创建一个游标来检索没有电子邮件地址的所有顾客，作为应用程序的组成部分，帮助操作人员找出空缺的电子邮件地址。

下面是创建此游标的DB2、MariaDB、MySQL和SQL Server版本：

输入▼

```
DECLARE CustCursor CURSOR
FOR
SELECT * FROM Customers
WHERE cust_email IS NULL
```

下面是Oracle和PostgreSQL版本：

输入▼

```
DECLARE CURSOR CustCursor
IS
SELECT * FROM Customers
WHERE cust_email IS NULL
```

分析▼
在上面两个版本中，DECLARE语句用来定义和命名游标，这里为CustCursor。SELECT语句定义一个包含没有电子邮件地址（NULL值）的所有顾客的游标。

定义游标之后，就可以打开它了。

21.2.2 使用游标

使用OPEN CURSOR语句打开游标，这条语句很简单，在大多数DBMS中的语法相同：

输入▼

```
OPEN CURSOR CustCursor
```

分析▼
在处理OPEN CURSOR语句时，执行查询，存储检索出的数据以供浏览和滚动。

现在可以用FETCH语句访问游标数据了。FETCH指出要检索哪些行，从何处检索它们以及将它们放于何处（如变量名）。第一个例子使用Oracle语法从游标中检索一行（第一行）：

输入▼

```
DECLARE TYPE CustCursor IS REF CURSOR
```

```
    RETURN Customers%ROWTYPE;
DECLARE CustRecord Customers%ROWTYPE
BEGIN
    OPEN CustCursor;
    FETCH CustCursor INTO CustRecord;
    CLOSE CustCursor;
END;
```

分析▼
在这个例子中，FETCH用来检索当前行（自动从第一行开始），放到声明的变量CustRecord中。对于检索出来的数据不做任何处理。

下一个例子（也使用Oracle语法）中，从第一行到最后一行，对检索出来的数据进行循环：

输入▼

```
DECLARE TYPE CustCursor IS REF CURSOR
    RETURN Customers%ROWTYPE;
DECLARE CustRecord Customers%ROWTYPE
BEGIN
    OPEN CustCursor;
    LOOP
    FETCH CustCursor INTO CustRecord;
    EXIT WHEN CustCursor%NOTFOUND;
    ...
    END LOOP;
    CLOSE CustCursor;
END;
```

分析▼
与前一个例子一样，这个例子使用FETCH检索当前行，放到一个名为CustRecord的变量中。但不一样的是，这里的FETCH位于LOOP内，因此它反复执行。代码EXIT WHEN CustCursor%NOTFOUND使在取不出更多的行时终止处理（退出循环）。这个例子也没有做实际的处理，实际例子中可用具体的处理代码替换占位符...。

下面是另一个例子，这次使用Microsoft SQL Server语法：

输入▼

```
DECLARE @cust_id CHAR(10),
```

```
        @cust_name CHAR(50),
        @cust_address CHAR(50),
        @cust_city CHAR(50),
        @cust_state CHAR(5),
        @cust_zip CHAR(10),
        @cust_country CHAR(50),
        @cust_contact CHAR(50),
        @cust_email CHAR(255)
OPEN CustCursor
FETCH NEXT FROM CustCursor
    INTO @cust_id, @cust_name, @cust_address,
         @cust_city, @cust_state, @cust_zip,
         @cust_country, @cust_contact, @cust_email
WHILE @@FETCH_STATUS = 0
BEGIN

FETCH NEXT FROM CustCursor
        INTO @cust_id, @cust_name, @cust_address,
             @cust_city, @cust_state, @cust_zip,
             @cust_country, @cust_contact, @cust_email
...
END
CLOSE CustCursor
```

分析▼

在此例中，为每个检索出的列声明一个变量，FETCH语句检索一行并保存值到这些变量中。使用WHILE循环处理每一行，条件WHILE @@FETCH_STATUS = 0在取不出更多的行时终止处理（退出循环）。这个例子也不进行具体的处理，实际代码中，应该用具体的处理代码替换其中的...占位符。

21.2.3　关闭游标

如前面几个例子所述，游标在使用完毕时需要关闭。此外，SQL Server等DBMS要求明确释放游标所占用的资源。下面是DB2、Oracle和PostgreSQL的语法：

输入▼

```
CLOSE CustCursor
```

下面是Microsoft SQL Server的版本：

输入▼

```
CLOSE CustCursor
DEALLOCATE CURSOR CustCursor
```

分析▼
CLOSE语句用来关闭游标。一旦游标关闭，如果不再次打开，将不能使用。第二次使用它时不需要再声明，只需用OPEN打开它即可。

21.3　小结

我们在本课讲授了什么是游标，为什么使用游标。你使用的DBMS可能会提供某种形式的游标，以及这里没有提及的功能。更详细的内容请参阅具体的DBMS文档。

第22课　高级SQL特性

这一课介绍SQL所涉及的几个高级数据处理特性：约束、索引和触发器。

22.1　约束

SQL已经改进过多个版本，成为非常完善和强大的语言。许多强有力的特性给用户提供了高级的数据处理技术，如约束。

关联表和引用完整性已经在前面讨论过几次。正如所述，关系数据库存储分解为多个表的数据，每个表存储相应的数据。利用键来建立从一个表到另一个表的引用（由此产生了术语引用完整性（referential integrity））。

正确地进行关系数据库设计，需要一种方法保证只在表中插入合法数据。例如，如果`Orders`表存储订单信息，`OrderItems`表存储订单详细内容，应该保证`OrderItems`中引用的任何订单ID都存在于`Orders`中。类似地，在`Orders`表中引用的任意顾客必须存在于`Customers`表中。

虽然可以在插入新行时进行检查（在另一个表上执行`SELECT`，以保证所有值合法并存在），但最好不要这样做，原因如下：

- 如果在客户端层面上实施数据库完整性规则，则每个客户端都要被迫实施这些规则，一定会有一些客户端不实施这些规则。
- 在执行`UPDATE`和`DELETE`操作时，也必须实施这些规则。
- 执行客户端检查是非常耗时的，而DBMS执行这些检查会相对高效。

> 约束（**constraint**）
> 管理如何插入或处理数据库数据的规则。

DBMS通过在数据库表上施加约束来实施引用完整性。大多数约束是在表定义中定义的，如第17课所述，用`CREATE TABLE`或`ALTER TABLE`语句。

> 注意：具体**DBMS**的约束

> 有几种不同类型的约束，每个DBMS都提供自己的支持。因此，这里给出的例子在不同的DBMS上可能有不同的反应。在进行试验之前，请参阅具体的DBMS文档。

22.1.1　主键

我们在第1课简单提过主键。主键是一种特殊的约束，用来保证一列（或一组列）中的值是唯一的，而且永不改动。换句话说，表中的一列（或多个列）的值唯一标识表中的每一行。这方便了直接或交互地处理表中的行。没有主键，要安全地UPDATE或DELETE特定行而不影响其他行会非常困难。

表中任意列只要满足以下条件，都可以用于主键：

- 任意两行的主键值都不相同。
- 每行都具有一个主键值（即列中不允许NULL值）。
- 包含主键值的列从不修改或更新。（大多数DBMS不允许这么做，但如果你使用的DBMS允许这样做，好吧，千万别！）
- 主键值不能重用。如果从表中删除某一行，其主键值不分配给新行。

一种定义主键的方法是创建它，如下所示：

输入▼

```
CREATE TABLE Vendors
(
vend_id         CHAR(10)      NOT NULL PRIMARY KEY,
vend_name       CHAR(50)      NOT NULL,
vend_address    CHAR(50)      NULL,
vend_city       CHAR(50)      NULL,
vend_state      CHAR(5)       NULL,
vend_zip        CHAR(10)      NULL,
vend_country    CHAR(50)      NULL
);
```

分析▼
在此例子中，给表的vend_id列定义添加关键字PRIMARY KEY，使其成为主键。

输入▼

```
ALTER TABLE Vendors
ADD CONSTRAINT PRIMARY KEY (vend_id);
```

分析▼

这里定义相同的列为主键，但使用的是CONSTRAINT语法。此语法也可以用于CREATE TABLE和ALTER TABLE语句。

> 说明：**SQLite中的键**
> SQLite不允许使用ALTER TABLE定义键，要求在初始的CREATE TABLE语句中定义它们。

22.1.2 外键

外键是表中的一列，其值必须列在另一表的主键中。外键是保证引用完整性的极其重要部分。我们举个例子来理解外键。

Orders表将录入到系统的每个订单作为一行包含其中。顾客信息存储在Customers表中。Orders表中的订单通过顾客ID与Customers表中的特定行相关联。顾客ID为Customers表的主键，每个顾客都有唯一的ID。订单号为Orders表的主键，每个订单都有唯一的订单号。

Orders表中顾客ID列的值不一定是唯一的。如果某个顾客有多个订单，则有多个行具有相同的顾客ID（虽然每个订单都有不同的订单号）。同时，Orders表中顾客ID列的合法值为Customers表中顾客的ID。

这就是外键的作用。在这个例子中，在Orders的顾客ID列上定义了一个外键，因此该列只能接受Customers表的主键值。

下面是定义这个外键的方法：

输入▼

```
CREATE TABLE Orders
(
    order_num      INTEGER     NOT NULL PRIMARY KEY,
    order_date     DATETIME    NOT NULL,
    cust_id        CHAR(10)    NOT NULL REFERENCES Customers(c
);
```

分析▼
其中的表定义使用了REFERENCES关键字，它表示cust_id中的任何值都必须是Customers表的cust_id中的值。

相同的工作也可以在ALTER TABLE语句中用CONSTRAINT语法来完成：

输入▼

```
ALTER TABLE Orders
ADD CONSTRAINT
FOREIGN KEY (cust_id) REFERENCES Customers (cust_id)
```

> 提示：外键有助防止意外删除
> 如第6课所述，除帮助保证引用完整性外，外键还有另一个重要作用。在定义外键后，DBMS不允许删除在另一个表中具有关联行的行。例如，不能删除关联订单的顾客。删除该顾客的唯一方法是首先删除相关的订单（这表示还要删除相关的订单项）。由于需要一系列的删除，因而利用外键可以防止意外删除数据。
>
> 有的DBMS支持称为级联删除（cascading delete）的特性。如果启用，该特性在从一个表中删除行时删除所有相关的数据。例如，如果启用级联删除并且从Customers表中删除某个顾客，则任何关联的订单行也会被自动删除。

22.1.3　唯一约束

唯一约束用来保证一列（或一组列）中的数据是唯一的。它们类似于主键，但存在以下重要区别。

- 表可包含多个唯一约束，但每个表只允许一个主键。
- 唯一约束列可包含NULL值。
- 唯一约束列可修改或更新。
- 唯一约束列的值可重复使用。
- 与主键不一样，唯一约束不能用来定义外键。

employees表是一个使用约束的例子。每个雇员都有唯一的社会安全号，但我们并不想用它作主键，因为它太长（而且我们也不想使该信息容易利用）。因此，每个雇员除了其社会安全号外还有唯一的雇员ID（主键）。

雇员ID是主键，可以确定它是唯一的。你可能还想使DBMS保证每个社会安全号也是唯一的（保证输入错误不会导致使用他人号码）。可以通过在社会安全号列上定义UNIQUE约束做到。

唯一约束的语法类似于其他约束的语法。唯一约束既可以用UNIQUE关键字在表定义中定义，也可以用单独的CONSTRAINT定义。

22.1.4 检查约束

检查约束用来保证一列（或一组列）中的数据满足一组指定的条件。检查约束的常见用途有以下几点。

- 检查最小或最大值。例如，防止0个物品的订单（即使0是合法的数）。
- 指定范围。例如，保证发货日期大于等于今天的日期，但不超过今天起一年后的日期。
- 只允许特定的值。例如，在性别字段中只允许M或F。

换句话说，第1课介绍的数据类型限制了列中可保存的数据的类型。检查约束在数据类型内又做了进一步的限制，这些限制极其重要，可以确保插入数据库的数据正是你想要的数据。不需要依赖于客户端应用程序或用户来保证正确获取它，DBMS本身将会拒绝任何无效的数据。

下面的例子对OrderItems表施加了检查约束，它保证所有物品的数量大于0：

输入▼

```
CREATE TABLE OrderItems
(
    order_num      INTEGER     NOT NULL,
    order_item     INTEGER     NOT NULL,
    prod_id        CHAR(10)    NOT NULL,
    quantity       INTEGER     NOT NULL CHECK (quantity > 0),
    item_price     MONEY       NOT NULL
);
```

分析▼
利用这个约束，任何插入（或更新）的行都会被检查，保证quantity大于0。

检查名为gender的列只包含M或F，可编写如下的ALTER TABLE语句：

输入▼

```
ADD CONSTRAINT CHECK (gender LIKE '[MF]')
```

> 提示：用户定义数据类型
> 有的DBMS允许用户定义自己的数据类型。它们是定义检查约束（或其他约束）的基本简单数据类型。例如，你可以定义自己的名为gender的数据类型，它是单字符的文本数据类型，带限制其值为M或F（对于未知值或许还允许NULL）的检查约束。然后，可以将此数据类型用于表的定义。定制数据类型的优点是只需施加约束一次（在数据类型定义中），而每当使用该数据类型时，都会自动应用这些约束。请查阅相应的DBMS文档，看它是否支持自定义数据类型。

22.2　索引

索引用来排序数据以加快搜索和排序操作的速度。想像一本书后的索引（如本书后的索引），可以帮助你理解数据库的索引。

假如要找出本书中所有的“数据类型”这个词，简单的办法是从第1页开始，浏览每一行。虽然这样做可以完成任务，但显然不是一种好的办法。浏览少数几页文字可能还行，但以这种方式浏览整部书就不可行了。随着要搜索的页数不断增加，找出所需词汇的时间也会增加。

这就是书籍要有索引的原因。索引按字母顺序列出词汇及其在书中的位置。为了搜索“数据类型”一词，可在索引中找出该词，确定它出现在哪些页中。然后再翻到这些页，找出“数据类型”一词。

使索引有用的因素是什么？很简单，就是恰当的排序。找出书中词汇的困难不在于必须进行多少搜索，而在于书的内容没有按词汇排序。如果书的内容像字典一样排序，则索引没有必要（因此字典就没有索引）。

数据库索引的作用也一样。主键数据总是排序的，这是DBMS的工作。因此，按主键检索特定行总是一种快速有效的操作。

但是，搜索其他列中的值通常效率不高。例如，如果想搜索住在某个

州的客户，怎么办？因为表数据并未按州排序，DBMS必须读出表中所有行（从第一行开始），看其是否匹配。这就像要从没有索引的书中找出词汇一样。

解决方法是使用索引。可以在一个或多个列上定义索引，使DBMS保存其内容的一个排过序的列表。在定义了索引后，DBMS以使用书的索引类似的方法使用它。DBMS搜索排过序的索引，找出匹配的位置，然后检索这些行。

在开始创建索引前，应该记住以下内容：

- 索引改善检索操作的性能，但降低了数据插入、修改和删除的性能。在执行这些操作时，DBMS必须动态地更新索引。
- 索引数据可能要占用大量的存储空间。
- 并非所有数据都适合做索引。取值不多的数据（如州）不如具有更多可能值的数据（如姓或名），能通过索引得到那么多的好处。
- 索引用于数据过滤和数据排序。如果你经常以某种特定的顺序排序数据，则该数据可能适合做索引。
- 可以在索引中定义多个列（例如，州加上城市）。这样的索引仅在以州加城市的顺序排序时有用。如果想按城市排序，则这种索引没有用处。

没有严格的规则要求什么应该索引，何时索引。大多数DBMS提供了可用来确定索引效率的实用程序，应该经常使用这些实用程序。

索引用`CREATE INDEX`语句创建（不同DBMS创建索引的语句变化很大）。下面的语句在`Products`表的产品名列上创建一个简单的索引：

输入▼

```
CREATE INDEX prod_name_ind
ON Products (prod_name);
```

分析▼
索引必须唯一命名。这里的索引名`prod_name_ind`在关键字`CREATE INDEX`之后定义。`ON`用来指定被索引的表，而索引中包含的列（此例中仅有一列）在表名后的圆括号中给出。

> 提示：检查索引

> 索引的效率随表数据的增加或改变而变化。许多数据库管理员发现，过去创建的某个理想的索引经过几个月的数据处理后可能变得不再理想了。最好定期检查索引，并根据需要对索引进行调整。

22.3 触发器

触发器是特殊的存储过程，它在特定的数据库活动发生时自动执行。触发器可以与特定表上的INSERT、UPDATE和DELETE操作（或组合）相关联。

与存储过程不一样（存储过程只是简单的存储SQL语句），触发器与单个的表相关联。与Orders表上的INSERT操作相关联的触发器只在Orders表中插入行时执行。类似地，Customers表上的INSERT和UPDATE操作的触发器只在表上出现这些操作时执行。

触发器内的代码具有以下数据的访问权：

- INSERT操作中的所有新数据；
- UPDATE操作中的所有新数据和旧数据；
- DELETE操作中删除的数据。

根据所使用的DBMS的不同，触发器可在特定操作执行之前或之后执行。

下面是触发器的一些常见用途。

- 保证数据一致。例如，在INSERT或UPDATE操作中将所有州名转换为大写。
- 基于某个表的变动在其他表上执行活动。例如，每当更新或删除一行时将审计跟踪记录写入某个日志表。
- 进行额外的验证并根据需要回退数据。例如，保证某个顾客的可用资金不超限定，如果已经超出，则阻塞插入。
- 计算计算列的值或更新时间戳。

读者可能已经注意到了，不同DBMS的触发器创建语法差异很大，更详细的信息请参阅相应的文档。

下面的例子创建一个触发器，它对所有INSERT和UPDATE操作，将Customers表中的cust_state列转换为大写。

这是本例子的SQL Server版本：

输入▼

```
CREATE TRIGGER customer_state
ON Customers
FOR INSERT, UPDATE
AS
UPDATE Customers
SET cust_state = Upper(cust_state)
WHERE Customers.cust_id = inserted.cust_id;
```

这是本例子的Oracle和PostgreSQL的版本：

输入▼

```
CREATE TRIGGER customer_state
AFTER INSERT OR UPDATE
FOR EACH ROW
BEGIN
UPDATE Customers
SET cust_state = Upper(cust_state)
WHERE Customers.cust_id = :OLD.cust_id
END;
```

> 提示：约束比触发器更快
> 一般来说，约束的处理比触发器快，因此在可能的时候，应该尽量使用约束。

22.4 数据库安全

对于组织来说，没有什么比它的数据更重要了，因此应该保护这些数据，使其不被偷盗或任意浏览。当然，数据也必须允许需要访问它的用户访问，因此大多数DBMS都给管理员提供了管理机制，利用管理机制授予或限制对数据的访问。

任何安全系统的基础都是用户授权和身份确认。这是一种处理，通过这种处理对用户进行确认，保证他是有权用户，允许执行他要执行的操作。有的DBMS为此结合使用了操作系统的安全措施，而有的维护自己的用户及密码列表，还有一些结合使用外部目录服务服务器。

一般说来，需要保护的操作有：

- 对数据库管理功能（创建表、更改或删除已存在的表等）的访问；
- 对特定数据库或表的访问；
- 访问的类型（只读、对特定列的访问等）；
- 仅通过视图或存储过程对表进行访问；
- 创建多层次的安全措施，从而允许多种基于登录的访问和控制；
- 限制管理用户账号的能力。

安全性使用SQL的GRANT和REVOKE语句来管理，不过，大多数DBMS提供了交互式的管理实用程序，这些实用程序在内部使用GRANT和REVOKE语句。

22.5　小结

本课讲授如何使用SQL的一些高级特性。约束是实施引用完整性的重要部分，索引可改善数据检索的性能，触发器可以用来执行运行前后的处理，安全选项可用来管理数据访问。不同的DBMS可能会以不同的形式提供这些特性，更详细的信息请参阅具体的DBMS文档。

附录A　样例表脚本

编写SQL语句需要良好地理解基本数据库设计。如果不知道什么信息存放在什么表中，表与表之间如何互相关联，行中数据如何分解，那么要编写高效的SQL是不可能的。

强烈建议读者实际练习本书的每个例子。所有课都共同使用了一组数据文件。为帮助你更好地理解这些例子、学好各课内容，本附录描述了所用的表、表之间的关系以及如何创建（或获得）它们。

A.1　样例表

本书中所用的表是一个假想玩具经销商使用的订单录入系统的组成部分。这些表用来完成以下几项任务：

- 管理供应商；
- 管理产品目录；
- 管理顾客列表；
- 录入顾客订单。

完成它们需要5个表（它们作为一个关系数据库设计的组成部分紧密关联）。以下各节给出每个表的描述。

> 说明：简化的例子
> 这里使用的表不完整，现实世界中的订单录入系统还会记录这里所没有的大量数据（如工资和记账信息、发货追踪信息等）。不过，这些表确实示范了现实世界中你将遇到的各种数据的组织和关系。读者可以将这些技术用于自己的数据库。

表的描述

下面介绍5个表及每个表内的列名。

1．`Vendors`表

`Vendors`表存储销售产品的供应商。每个供应商在这个表中有一个记录，供应商ID列（`vend_id`）用于进行产品与供应商的匹配。

表A-1 **Vendors**表的列

列	说 明
vend_id	唯一的供应商ID
vend_name	供应商名
vend_address	供应商的地址
vend_city	供应商所在城市
vend_state	供应商所在州
vend_zip	供应商地址邮政编码
vend_country	供应商所在国家

- 所有表都应该有主键。这个表应该用vend_id作为其主键。

2. **Products**表

Products表包含产品目录，每行一个产品。每个产品有唯一的ID（prod_id列），并且借助vend_id（供应商的唯一ID）与供应商相关联。

表A-2 **Products**表的列

列	说 明
prod_id	唯一的产品ID
vend_id	产品供应商ID（关联到Vendors表的vend_id）
prod_name	产品名
prod_price	产品价格
prod_desc	产品描述

- 所有表都应该有主键。这个表应该用prod_id作为其主键。
- 为实施引用完整性，应该在vend_id上定义一个外键，关联到Vendors的vend_id列。

3. **Customers**表

Customers表存储所有顾客信息。每个顾客有唯一的ID（cust_id列）。

表A-3 **Customers**表的列

列	说 明
cust_id	唯一的顾客ID
cust_name	顾客名

cust_address	顾客的地址
cust_city	顾客所在城市
cust_state	顾客所在州
cust_zip	顾客地址邮政编码
cust_country	顾客所在国家
cust_contact	顾客的联系名
cust_email	顾客的电子邮件地址

- 所有表都应该有主键。这个表应该用cust_id作为它的主键。

4. **Orders**表

Orders表存储顾客订单（不是订单细节）。每个订单唯一编号（order_num列）。Orders表按cust_id列（关联到Customers表的顾客唯一ID）关联到相应的顾客。

表A-4 **Orders**表的列

列	说　　明
order_num	唯一的订单号
order_date	订单日期
cust_id	订单顾客ID（关联到Customers表的cust_id）

- 所有表都应该有主键。这个表应该用order_num作为其主键。
- 为实施引用完整性，应该在cust_id上定义一个外键，关联到Customers的cust_id列。

5. **OrderItems**表

OrderItems表存储每个订单中的实际物品，每个订单的每个物品一行。对于Orders表的每一行，在OrderItems表中有一行或多行。每个订单物品由订单号加订单物品（第一个物品、第二个物品等）唯一标识。订单物品用order_num列（关联到Orders表中订单的唯一ID）与其相应的订单相关联。此外，每个订单物品包含该物品的产品ID（把物品关联到Products表）。

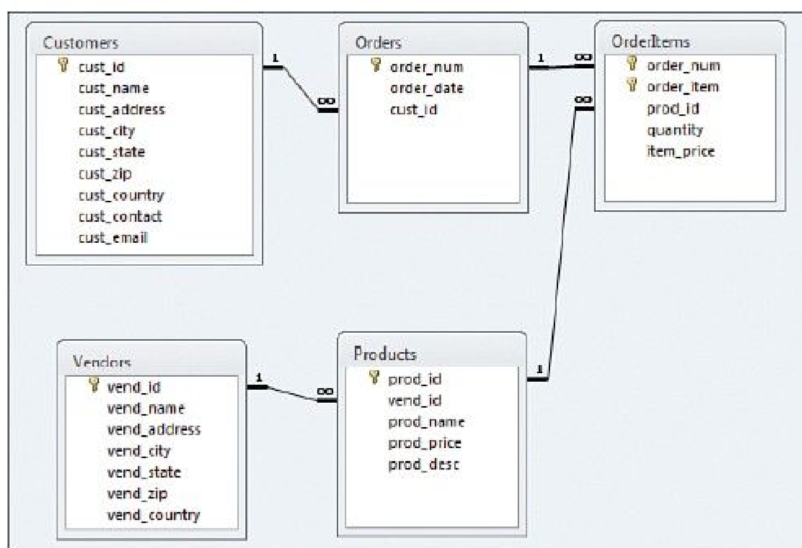
表A-5 **OrderItems**表的列

列	说　　明
order_num	订单号（关联到Orders表的order_num）
order_item	订单物品号（订单内的顺序）
prod_id	产品ID（关联到Products表的prod_id）

quantity	物品数量
item_price	物品价格

- 所有表都应该有主键。这个表应该用order_num和order_item作为其主键。
- 为实施引用完整性，应该在order_num和prod_id上定义外键，关联order_num到Orders的order_num列，关联prod_id到Products的prod_id列。

数据库管理员通常使用关系图来说明数据库表的关联方式。要记住，正如上面表描述提到的，外键定义了这些关系。图A-1是本附录描述的五个表的关系图。



图**A-1** 样例表关系图

A.2 获得样例表

学习各个例子，需要一组填充了数据的表。所需要获得和运行的东西都可以在本书网页http://www.forta.com/books/0672336073/找到。

A.2.1 下载可供使用的数据文件

可从上述URL下载一个填充了数据的如下格式的文件：

- Apache Open Office Base
- Microsoft Access（2000和2007）
- SQLite

如果使用这些文件，不需要执行任何SQL创建和填充脚本。

A.2.2　下载DBMS SQL脚本

大多数DBMS以不自己完成文件分布的格式存储数据（如Access、Open Office Base和SQLite那样）。对于这些DBMS，可以从上述URL下载SQL脚本。对于每个DBMS，有两个文件：

- create.text包含创建5个数据库表（包括定义所有主键和外键约束）的SQL语句。
- populate.txt包含用来填充这些表的SQL `INSERT`语句。

这些文件中的SQL语句依赖于具体的DBMS，因此应该执行适合于你的DBMS的那个。这些脚本为方便读者而提供，作者对执行它们万一引起的问题不承担任何责任。

在本书付印时，有以下脚本可供使用：

- IBM DB2；
- Microsoft SQL Server（包括Microsoft SQL Server Express）；
- MariaDB
- MySQL；
- Oracle（包括Oracle Express）；
- PostgreSQL。

适用于其他DBMS的脚本可能会根据需要或请求而增加。

附录B提供了在几个流行环境中执行脚本的说明。

> 说明：创建，然后填充
> 必须在执行表填充脚本前执行表创建脚本。应该检查这些脚本返回的错误。如果创建脚本失败，则应该在继续表填充前解决存在的问题。

> 说明：具体**DBMS**的设置指令
> 用于设置DBMS的具体步骤依使用的DBMS有很大不同。从本书网页下载脚本或数据库时，你会看到README文件，它提供了针对特定DBMS的具体设置和安装步骤。

附录B　流行的应用程序

正如第1课所述，SQL不是一个应用，而是一种语言。要学习本书中的例子，你需要一个支持SQL语句执行的应用程序。

本附录描述了在几个最常用的应用中执行SQL语句的步骤。

你可以使用下面列出的任何一个应用或其他应用来测试和试验SQL代码。那么，究竟应该使用哪个呢？

- 许多DBMS具有自己的客户端实用程序，它们通常是很好的学习出发点。但有些没有直观的用户界面。
- 如果你是一位Web开发人员，那么可以使用任何服务器端Web编程语言，包括ASP.NET、ColdFusion、Java、JSP、PHP、Python和Ruby on Rails。
- 有很多第三方工具和实用程序，本书网页http://www.forta.com/books/0672336073/上有其中的一些的链接。

B.1　使用Apache Open Office Base

Apache Open Office Base是一个基于Java的开源客户端数据库应用。Open Office Base查询可以使用SQL直接编写。为此，要执行如下步骤。

1. 在Open Office Base中打开数据库。
2. 选择左侧Database面板中的Queries。
3. 在Tasks面板，点击Create Query In SQL View以显示Query Design窗口。
4. 在大文本框（整个窗口都是文本框）中输入SQL语句。
5. 执行SQL语句，点击Run Query按钮（有文档和绿色对钩的按钮）。也可以按F5或从Edit菜单选择Run Query，执行SQL语句。

B.2　使用Adobe ColdFusion

Adobe ColdFusion是一个Web应用开发平台。ColdFusion使用一个基于标签的语言来创建脚本。为测试SQL，需要创建一个可从Web浏览

器调用执行的简单页面。执行以下步骤。

1. 在从ColdFusion代码内使用任意数据库之前，必须定义一个数据源。ColdFusion Administrator程序提供了一个定义数据源的基于Web的界面（如果需要帮助，请参阅ColdFusion文档）。
2. 创建新的ColdFusion页（用CFM扩展）。
3. 使用CFML `<CFQUERY>`和`</CFQUERY>`标签创建一个查询块。用`NAME`属性命名它并在`DATASOURCE`属性中定义Data Source。
4. 在`<CFQUERY>`和`</CFQUERY>`标签之间输入SQL语句。
5. 使用`<CFDUMP>`或`<CFOUTPUT>`循环显示查询结果。
6. 在Web服务器根目录下的任意可执行目录中保存此页面。
7. 通过从Web浏览器中调用此页面执行它。

B.3　使用IBM DB2

IBM的DB2是一个强有力的高端多平台DBMS。它带有一整套可用来执行SQL语句的客户端工具。下面的说明使用基于Java的Control Center实用程序，因为它是最简单且最通用的绑定应用程序。

1. 运行Control Center。
2. 左侧的Object View列出了所有可用的数据库，展开All Databases，选择你需要的数据库。
3. 在选择的数据库上点击右键并选择Query，或者（虽然它被选中）从Selected菜单选择Query。
4. 在上面的框中输入SQL语句。
5. 点击Execute按钮（有绿色朝右箭头的按钮），执行这个脚本。
6. 下面的窗口会显示状态信息，切换到Query Results标签页，可以以网格形式显示结果。

B.4　使用MariaDB

MariaDB没有自己的客户端实用程序，而是使用MySQL的客户端实用程序（完全兼容）。请参阅B.10节。

B.5　使用Microsoft Access

Microsoft Access通常用来交互式地创建和管理数据库，并且交互式地处理数据，Access给出Query Designer，可用来交互式地建立SQL

语句。Query Designer的一个相当有用的特性是，它也允许给出直接执行的SQL。为使用这个特性，进行如下操作。

1. 运行Microsoft Access。它将提示你打开（或创建）数据库。打开要使用的数据库。
2. 在Database窗口中选择Queries，然后单击New按钮并选择Design View。
3. 出现Show Table对话框，关闭此窗口，不选择任何表。
4. 从View菜单选择SQL View显示Query窗口。
5. 在Query窗口中输入要执行的SQL语句。
6. 单击Run按钮（有红惊叹号的那个）执行SQL语句。这将切换视图到Datasheet View（它将在网格中显示结果）。
7. 根据需要在SQL View和Datasheet View间切换（为改动SQL语句，需返回到SQL View）。还可以使用Design View交互式地建立SQL语句。

B.6　使用Microsoft ASP

Microsoft ASP是创建基于Web应用的脚本平台。在ASP页内测试SQL语句，必须创建一个通过从Web浏览器调用来执行的页面。下面是在ASP页面内执行SQL语句所需的步骤：

1. ASP使用ODBC与数据库交互，因此在继续进行之前必须给出ODBC数据源（请参阅本附录最后面的内容）。
2. 用任意文本编辑器创建新ASP页面（带ASP扩展）。
3. 使用`Server.CreateObject`创建`ADODB.Connection`对象的一个实例。
4. 使用`Open`方法打开所需的ODBC数据源。
5. 将SQL语句传递到一个对`Execute`方法的调用。`Execute`方法返回结果集。使用`Set`命令将返回的结果保存到结果集。
6. 为显示结果，应该对检索出的数据应用`<% Do While NOT EOF %>`循环。
7. 在Web服务器根目录下的任意可执行目录中保存此页面。
8. 通过从Web浏览器调用，执行此页面。

B.7　使用Microsoft ASP.NET

Microsoft ASP.NET是一个使用.NET框架，创建基于Web应用的脚本平台。在ASP.NET页面内测试SQL语句，必须创建一个可通过从浏览

器调用来执行的页面。有多种方法完成这项工作，下面是其中的一种方法。

1. 创建一个带.aspx扩展的新文件。
2. 用`SqlConnection()`或`OleDbConnection()`创建数据库连接。
3. 使用`SqlCommand()`或`OleDbCommand()`给DBMS传递语句。
4. 用`ExecuteReader`创建一个DataReader。
5. 对返回的读入器（reader）循环以获得返回值。
6. 在Web服务器根目录下的任意可执行目录中保存页面。
7. 通过从Web浏览器调用页面执行它。

B.8 使用Microsoft Query

Microsoft Query是一个独立的SQL查询工具，也是一个对ODBC数据源测试SQL语句的理想实用程序。大多数Windows安装程序不安装Microsoft Query，不过，它可以与其他Microsoft产品以及第三方产品一起选择安装。如果你的计算机上存在就可以使用。

> 提示：获得**MS-Query**
> MS-Query通常与Office等其他微软产品一起安装，虽然它只能在进行完整安装时安装。如果Start按钮上没有给出，可用Start Find在系统上查找它（不管你是否知道，一般都会给出它）。要查找的文件为MSQRY32.EXE或MSQUERY.EXE。

使用Microsoft Query，进行如下操作。

1. Microsoft Query使用ODBC与数据库打交道，因此在可以继续进行之前必须给出ODBC数据源（参见B.16的说明）。
2. 使用Microsoft Query之前，必须在计算机上安装它。请浏览Start按钮下的程序组找到它。
3. 从File菜单中，选择Execute SQL，显示Execute SQL窗。
4. 单击Data Sources按钮选择所要的ODBC数据源。如果所需的数据源未列出，单击Other寻找。选择了合适的数据源后，单击Use按钮。
5. 在SQL Statement框中输入SQL语句。
6. 单击Execute执行SQL语句并显示返回数据。

B.9 使用Microsoft SQL Server（包括

Microsoft SQL Server Express）

Microsoft SQL Server给出了一个称为SQL Server Management Studio的强大管理工具。这个工具用来管理数据库、管理安全、制作报表等各种工作，给出了编写和测试SQL语句的理想环境。以下说明如何使用SQL Server Management Studio。

1. 运行SQL Server Management Studio。
2. 如果需要，提供服务器和登录信息。
3. SQL Server Management Studio会展示多个面板。左侧的Object Explorer列出了所有的数据库及其详细信息，上面的工具条包含功能按钮，最大的文本区域用于输入SQL语句。
4. 工具条下方左侧的下拉选择框显示当前的数据库，也允许改变数据库。（做出选择在功能上等同于发出一条`USE`语句。）确保你使用的数据库是正确的，如果需要就切换数据库。
5. 在文本区域输入SQL语句，然后点击Execute Query按钮（有红色感叹号的），执行它（还可按F5或从Query菜单选择Execute）。
6. 结果将显示在SQL窗下的独立窗格中。
7. 单击查询屏幕底部的标签页，在查看数据与查看返回消息和信息之间切换。

B.10　使用MySQL

有两种方式使用MySQL。DBMS带有一个名为`mysql`的命令行实用程序。这是一个纯文本工具，通常作为MySQL安装程序的一部分来安装，用来执行任何SQL语句。另外，MySQL的创建者发布了一个名为MySQL Workbench的交互工具，通常需要独立下载和安装，所以它不会出现在其他安装程序中。学习MySQL时，强烈推荐使用它。

从命令行使用`mysql`，进行如下操作。

1. 输入`mysql`运行实用程序。根据如何定义安全性，可能需要使用`-u`和`-p`参数指定登录信息。
2. 在`mysql>`提示下输入`USE database`打开数据库，例如`USE tysql`就是打开`tysql`数据库。
3. 在`mysql>`提示下输入SQL语句，每条语句必须以分号（;）结束。结果将显示在屏幕上。
4. 为可能使用的命令列表输入`\h`，为状态信息输入`\s`（包括

MySQL版本信息）。
5. 输入`\q`退出`mysql`实用程序。

使用MySQL Workbench，进行如下操作。

1. 运行MySQL Workbench。
2. MySQL Workbench的最左侧列出了可用的MySQL数据库连接，允许你访问它们。点击任何连接就可以打开这个数据库；如果数据库没有在此列出，请选择New Connection。
3. 一旦连接，就会显示多个窗口。左侧的Object Browser列出了可用的数据库，中间是用于输入SQL语句的大的文本编辑器，结果或信息显示在下方。
4. 点击 + SQL按钮，打开新的SQL窗口。
5. 输入SQL语句后，点击Execute（有闪电图片的那个）运行SQL。结果显示在下面。

B.11　使用Oracle

Oracle有一大套工具和客户端。学习SQL时首选Oracle SQL Developer，它可以与MySQL DBMS一起安装，也可以独立下载并安装。下面介绍如何使用此工具。

1. 运行Oracle SQL Developer（Windows用户需要使用提供的批处理文件运行，不能通过应用本身运行）。
2. 在使用数据库前，你需要定义一个连接。这可以使用左侧Connections面板中的选项完成。
3. 连接完成后，使用SQL Worksheet标签页，在Query Builder屏幕中输入SQL语句。
4. 执行SQL语句，单击Execute按钮（带闪电图形）。结果将显示在下面的面板中。

B.12　使用Oracle Express

Oracle Express是一种强有力、易于使用的DBMS，有着非常直观、基于Web的用户界面。一旦安装了Oracle Express，就会有个Getting Started来启动Web页面管理，也可以使用该页面来运行SQL命令。为此，要按以下步骤操作。

1. 运行Oracle Express管理Web页面。

2. 出现提示，使用安装时的用户名和密码登录。
3. 登录后可以看到一系列图标，包括一个含单词SQL的屏幕图片。点击这个图标可以访问选项。
4. 第一个图标是SQL Commands，可用来输入SQL语句。（第二个图标是SQL Scripts，执行已经编写的SQL脚本时很有用，像创建和填充本书可用的脚本。）点击SQL Commands图标，打开SQL Commands窗口。
5. 在屏幕上方输入SQL语句。
6. 执行SQL查询，点击右上方的Run按钮。结果显示在SQL语句之下。

B.13　使用PHP

PHP是一个流行的Web脚本语言。它提供了用来连接各种数据库的函数和库，因此，用来执行SQL语句的代码可根据所用（以及如何访问）的DBMS而变化。因而，这里无法提供用于各种DBMS和各种情形的步骤，只提供一个使用MySQL的一般例子。关于如何连接到具体的DBMS的介绍，请参阅PHP文档。

1. 创建一个新的PHP页面（和一个PHP扩展）。
2. 使用适当的函数连接数据库。使用`mysql_connect()`连接MySQL数据库。
3. 将SQL语句传给适当的查询函数。针对MySQL，使用`mysql_query()`。
4. 返回一组结果，循环遍历它们以显示。
5. 将页面保存在Web服务器根目录下的任何可执行目录中。
6. 通过从Web浏览器调用来执行页面。

B.14　使用PostgreSQL

有两种方式使用PostgreSQL。DBMS带有一个名为`psql`的命令行实用程序，这是一个纯文本工具，通常与PostgreSQL一起安装，用来执行任意SQL语句。另外，有一个名为pgAdmin的交互工具，主要用于DBMS管理，可测试SQL语句。

使用`psql`，进行如下操作。

1. 输入`psql`运行此实用程序。加载一个特定的数据库，请在命令行上指定它，格式为：`psql database`（PostgreSQL不支持

USE命令）。
2. 在=>提示下输入要执行的SQL语句，每条语句以分号（;）结束。结果将显示在屏幕上。
3. 输入\?，可列出可能用到的命令。
4. 输入\h，可得到SQL帮助；输入\h statement（如\h SELECT），可得到特定SQL语句的帮助。
5. 输入\q退出psql实用程序。

使用pgAdmin，进行如下操作。

1. 运行pgAdmin（它也可能称为pgAdmin III）。
2. 如果出现提示，提供登录信息。
3. pgAdmin会在左侧列出数据库服务器。选择要使用的数据库，pgAdmin将连接它，并显示服务器信息。
4. 然后选择实际的数据库 （这样做，工具栏按钮即可使用了）。
5. 定位Execute Arbitrary SQL Queries按钮（是的，这确实是它的名字，它是那个包含单词SQL的放大镜图片）。
6. 将显示一个新的屏幕。确保要用的数据库列在了右上方的下拉选择框中。
7. 现在可以在屏幕上方的大文本框中输入SQL语句了。
8. 点击Execute Query按钮（有绿色向右箭头的那个），执行SQL语句。结果显示在下面。

B.15　使用SQLite

SQLite通常在其他语言和应用程序中使用，不作为单独的数据库使用。然而，SQLite确实有一个命令行实用程序，可以用来执行SQLite数据库的语句。SQLite命令行实用程序称为sqlite3（或Windows上的sqlite3.exe）。从命令行使用sqlite3，要进行如下操作。

1. 理想情况下，sqlite3和数据库可以放在同一文件夹下。（SQLite数据库包含在有.sqlite或.db扩展名的单独文件中，但事实上可以是任意扩展名，或者没有扩展名。）
2. 输入sqlite3 database.sqlite（用实际的数据库文件名替换database.sqlite）。
3. 可以看到sqlite>提示符，在此输入SQL语句。所有语句必须用分号（;）结束。
4. 默认情况下，sqlite3用列之间的管道字符显示返回的数据，且没有标题。要改变这种行为，输入.mode column并按回车，然后输入.header on并按回车。

5. 输入`.quit`并按回车，退出`sqlite3`。

B.16　配置ODBC数据源

上面描述的几个应用程序使用了ODBC进行数据库集成，因此，这里简要概述一下ODBC，以及配置ODBC数据源的指令。

ODBC是一个标准，能使客户端应用与不同的后端数据库或基础数据库引擎交互。使用ODBC，能够在一个客户端中编写代码，并使前述各种工具与几乎所有数据库或DBMS交互。

ODBC本身不是数据库，但它包装了数据库，使所有数据库以一致和清晰定义的方式工作。它利用具有两种主要功能的软件驱动程序实现这一点。首先，它们封装数据库的本身特性或特色，并对客户端隐藏它们。其次，它们提供一种常用的语言与这些数据库交互（在需要时进行转换）。ODBC所用的语言就是SQL。

ODBC客户端应用程序并不直接与数据库交互，而是与ODBC数据源交互。数据源是一个逻辑数据库，包括驱动程序（每种类型的数据库有自己的驱动程序）和如何连接到数据库的信息（文件路径、服务器名等）。

定义了ODBC数据源后，任何兼容ODBC的应用程序都可以使用这些数据源。ODBC数据源并不针对具体的应用程序，它们针对的是系统。

> 警告：**ODBC**的差别
> 存在许多不同的ODBC程序版本，因此不可能提供适用于所有版本的指令。在设置具体的数据源时，应该密切注意具体的提示。

ODBC数据源用Windows Control Panel的ODBC程序来定义。设置ODBC数据源，进行如下操作。

1. 打开Windows Control Panel的ODBC程序。
2. 大多数ODBC数据源应该设置为系统范围的数据源（相对于用户专用的数据源），因此，如果可以，应该选择System DSN。
3. 单击Add按钮添加新的数据源。
4. 选择要使用的驱动程序。通常有一组默认的驱动程序，支持主要的微软产品。你的系统上也可以安装其他驱动程序。必须选择一个与将要连接到的数据库类型相匹配的驱动程序。
5. 系统根据数据库或DBMS的类型，提示输入服务器名或文件路

径信息，以及可能的登录信息。根据要求提供这些信息，然后遵循其他提示创建数据源。

附录C　SQL语句的语法

为帮助读者在需要时找到相应语句的语法，本附录列出了最常使用的SQL语句的语法。每条语句以简要的描述开始，然后给出它的语法。为更方便查询，还标注了相应语句所在的课。

在阅读语句语法时，应该记住以下约定。

- |符号用来指出几个选择中的一个，因此，NULL | NOT NULL表示或者给出NULL或者给出NOT NULL。
- 包含在方括号中的关键字或子句（如[like this]）是可选的。
- 下面列出的语法几乎对所有DBMS都有效。关于具体语法可能变动的细节，建议读者参考自己的DBMS文档。

C.1　ALTER TABLE

ALTER TABLE用来更新已存在表的结构。为了创建新表，应该使用CREATE TABLE。详细信息，请参阅第17课。

输入▼

```
ALTER TABLE tablename
(
  ADD|DROP column datatype [NULL|NOT NULL] [CONSTRAINTS],
  ADD|DROP column datatype [NULL|NOT NULL] [CONSTRAINTS],
     ...
);
```

C.2　COMMIT

COMMIT用来将事务写入数据库。详细内容请参阅第20课。

输入▼

```
COMMIT [TRANSACTION];
```

C.3 CREATE INDEX

CREATE INDEX用于在一个或多个列上创建索引。详细内容请参阅第22课。

输入▼

```
CREATE INDEX indexname
ON tablename (column, ...);
```

C.4 CREATE PROCEDURE

CREATE PROCEDURE用于创建存储过程。详细内容请参阅第19课。正如所述，Oracle使用的语法稍有不同。

输入▼

```
CREATE PROCEDURE procedurename [parameters] [options]
AS
SQL statement;
```

C.5 CREATE TABLE

CREATE TABLE用于创建新数据库表。更新已经存在的表的结构，使用ALTER TABLE。详细内容请参阅第17课。

输入▼

```
CREATE TABLE tablename
(
    column datatype [NULL|NOT NULL] [CONSTRAINTS],
    column datatype [NULL|NOT NULL] [CONSTRAINTS],
       ...
);
```

C.6 CREATE VIEW

CREATE VIEW用来创建一个或多个表上的新视图。详细内容请参阅

第18课。

输入▼

```
CREATE VIEW viewname AS
SELECT columns, ...
FROM tables, ...
[WHERE ...]
[GROUP BY ...]
[HAVING ...];
```

C.7 DELETE

DELETE从表中删除一行或多行。详细内容请参阅第16课。

输入▼

```
DELETE FROM tablename
[WHERE ...];
```

C.8 DROP

DROP永久地删除数据库对象（表、视图、索引等）。详细内容请参阅第17、18课。

输入▼

```
DROP INDEX|PROCEDURE|TABLE|VIEW
indexname|procedurename|tablename|viewname;
```

C.9 INSERT

INSERT为表添加一行。详细内容请参阅第15课。

输入▼

```
INSERT INTO tablename [(columns, ...)]
VALUES(values, ...);
```

C.10 INSERT SELECT

INSERT SELECT将SELECT的结果插入到一个表。详细内容请参阅第15课。

输入▼

```
INSERT INTO tablename [(columns, ...)]
SELECT columns, ... FROM tablename, ...
[WHERE ...];
```

C.11 ROLLBACK

ROLLBACK用于撤销一个事务块。详细内容请参阅第20课。

输入▼

```
ROLLBACK [ TO savepointname];
```

或者：

输入▼

```
ROLLBACK TRANSACTION;
```

C.12 SELECT

SELECT用于从一个或多个表（视图）中检索数据。更多的基本信息，请参阅第2、3、4课（2～14课都与SELECT有关）。

输入▼

```
SELECT columnname, ...
FROM tablename, ...
[WHERE ...]
[UNION ...]
[GROUP BY ...]
[HAVING ...]
[ORDER BY ...];
```

C.13 UPDATE

UPDATE更新表中的一行或多行。详细内容请参阅第16课。

输入▼

```
UPDATE tablename
SET columnname = value, ...
[WHERE ...];
```

附录D　SQL数据类型

正如第1课所述，数据类型是定义列中可以存储什么数据以及该数据实际怎样存储的基本规则。

数据类型用于以下目的。

- 数据类型允许限制可存储在列中的数据。例如，数值数据类型列只能接受数值。
- 数据类型允许在内部更有效地存储数据。可以用一种比文本字符串更简洁的格式存储数值和日期时间值。
- 数据类型允许变换排序顺序。如果所有数据都作为字符串处理，则1位于10之前，而10又位于2之前（字符串以字典顺序排序，从左边开始比较，一次一个字符）。作为数值数据类型，数值才能正确排序。

在设计表时，应该特别重视所用的数据类型。使用错误的数据类型可能会严重影响应用程序的功能和性能。更改包含数据的列不是一件小事（而且这样做可能会导致数据丢失）。

本附录虽然不是关于数据类型及其如何使用的完整教材，但介绍了主要的数据类型、用途、兼容性等问题。

> 警告：任意两个**DBMS**都不是完全相同的
> 以前曾经说过，现在还需要再次提醒。不同DBMS的数据类型可能有很大的不同。在不同DBMS中，即使具有相同名称的数据类型也可能代表不同的东西。关于具体的DBMS支持何种数据类型以及如何支持的详细信息，请参阅具体的DBMS文档。

D.1　字符串数据类型

最常用的数据类型是字符串数据类型。它们存储字符串，如名字、地址、电话号码、邮政编码等。有两种基本的字符串类型，分别为定长字符串和变长字符串（参见表D-1）。

定长字符串接受长度固定的字符串，其长度是在创建表时指定的。例如，名字列可允许30个字符，而社会安全号列允许11个字符（允许的字符数目中包括两个破折号）。定长列不允许多于指定的字符数目。

它们分配的存储空间与指定的一样多。因此，如果字符串Ben存储到30个字符的名字字段，则存储的是30个字符，缺少的字符用空格填充，或根据需要补为NULL。

变长字符串存储任意长度的文本（其最大长度随不同的数据类型和DBMS而变化）。有些变长数据类型具有最小的定长，而有些则是完全变长的。不管是哪种，只有指定的数据得以保存（额外的数据不保存）。

既然变长数据类型这样灵活，为什么还要使用定长数据类型？答案是性能。DBMS处理定长列远比处理变长列快得多。此外，许多DBMS不允许对变长列（或一个列的可变部分）进行索引，这也会极大地影响性能（详细请参阅第22课）。

表**D-1** 串数据类型

数据类型	说　　明
CHAR	1～255个字符的定长字符串。它的长度必须在创建时规定
NCHAR	CHAR的特殊形式，用来支持多字节或Unicode字符（此类型的不同实现变化很大）
NVARCHAR	TEXT的特殊形式，用来支持多字节或Unicode字符（此类型的不同实现变化很大）
TEXT（也称为LONG、MEMO或VARCHAR）	变长文本

> 提示：使用引号
> 不管使用何种形式的字符串数据类型，字符串值都必须括在单引号内。

> 警告：当数值不是数值时
> 你可能会认为电话号码和邮政编码应该存储在数值字段中（数值字段只存储数值数据），但是这样做并不可取。如果在数值字段中存储邮政编码01234，则保存的将是数值1234，实际上丢失了一位数字。
>
> 需要遵守的基本规则是：如果数值是计算（求和、平均等）中使用的数值，则应该存储在数值数据类型列中；如果作为字符串（可能只包含数字）使用，则应该保存在字符串数据类型列中。

D.2　数值数据类型

数值数据类型存储数值。多数DBMS支持多种数值数据类型，每种存储的数值具有不同的取值范围。显然，支持的取值范围越大，所需存储空间越多。此外，有的数值数据类型支持使用十进制小数点（和小数），而有的则只支持整数。表D-2列出了常用的数值数据类型。并非所有DBMS都支持所列出的名称约定和描述。

表**D-2** 数值数据类型

数据类型	说　　明
BIT	单个二进制位值，或者为0或者为1，主要用于开/关标志
DECIMAL（或NUMERIC）	定点或精度可变的浮点值
FLOAT（或NUMBER）	浮点值
INT（或INTEGER）	4字节整数值，支持-2147483648～2147483647的数
REAL	4字节浮点值
SMALLINT	2字节整数值，支持-32768～32767的数
TINYINT	1字节整数值，支持0～255的数

> 提示：不使用引号
> 与字符串不一样，数值不应该括在引号内。

> 提示：货币数据类型
> 多数DBMS支持一种用来存储货币值的特殊数值数据类型。一般记为`MONEY`或`CURRENCY`，这些数据类型基本上是有特定取值范围的`DECIMAL`数据类型，更适合存储货币值。

D.3　日期和时间数据类型

所有DBMS都支持用来存储日期和时间值的数据类型（见表D-3）。与数值一样，多数DBMS都支持多种数据类型，每种具有不同的取值范围和精度。

表**D-3** 日期和时间数据类型

数据类型	说　　明

DATE	日期值
DATETIME（或TIMESTAMP）	日期时间值
SMALLDATETIME	日期时间值，精确到分（无秒或毫秒）
TIME	时间值

> 警告：指定日期
> 不存在所有DBMS都理解的定义日期的标准方法。多数实现都理解诸如`2015-12-30`或`Dec 30th, 2015`等格式，但即使这样，有的DBMS还是不理解它们。至于具体的DBMS能识别哪些日期格式，请参阅相应的文档。

> 提示：**ODBC**日期
> 因为每种DBMS都有自己特定的日期格式，所以ODBC创建了一种自己的格式，在使用ODBC时对每种数据库都起作用。ODBC格式对于日期类似于`{d '2005-12-30'}`，对于时间类似于`{t '21:46:29'}`，而对于日期时间类似于`{ts '2005-12-30 21:46:29'}`。如果通过ODBC使用SQL，应该以这种方式格式化日期和时间。

D.4　二进制数据类型

二进制数据类型是最不具有兼容性（幸运的是，也是最少使用）的数据类型。与迄今为止介绍的所有数据类型（它们具有特定的用途）不一样，二进制数据类型可包含任何数据，甚至可包含二进制信息，如图像、多媒体、字处理文档等（参见表D-4）。

表**D-4** 二进制数据类型

数据类型	说　　明
BINARY	定长二进制数据（最大长度从255字节到8000字节，有赖于具体的实现）
LONG RAW	变长二进制数据，最长2 GB
RAW（某些实现为BINARY）	定长二进制数据，最多255字节
VARBINARY	变长二进制数据（最大长度一般在255字节到8000字节间变化，依赖于具体的实现）

> 警告：数据类型对比

如果你想看一个数据库比较的实际例子，请考虑本书中用来建立样例表的表创建脚本（参看附录A）。通过比较这些用于不同DBMS的脚本，可看到数据类型匹配是一项多么复杂的任务。

附录E　SQL保留字

SQL是由关键字组成的语言，关键字是一些用于执行SQL操作的特殊词汇。在命名数据库、表、列和其他数据库对象时，一定不要使用这些关键字。因此，这些关键字是一定要保留的。

本附录列出主要DBMS中最常用的保留字。请注意以下几点。

- 关键字随不同的DBMS而变化，并非下面的所有关键字都被所有DBMS采用。
- 许多DBMS扩展了SQL保留字，使其包含专门用于实现的术语。多数DBMS专用的关键字未列在下面。
- 为保证以后的兼容性和可移植性，应避免使用这些保留字，即使它们不是你使用的DBMS的保留字。

ABORT	ABSOLUTE
ACTIVE	ADD
ALL	ALLOCATE
ANALYZE	AND
ARE	AS
ASCENDING	ASSERTION
AUTHORIZATION	AUTO
AUTOINC	AVG
BEFORE	BEGIN
BIGINT	BINARY
BLOB	BOOLEAN
BREAK	BROWSE
BY	BYTES
CALL	CASCADE
CASE	CAST
CHANGE	CHAR
CHARACTER_LENGTH	CHECK
CLOSE	CLUSTER
COALESCE	COLLATE
COLUMNS	COMMENT
COMMITTED	COMPUTE
CONDITIONAL	CONFIRM
CONNECTION	CONSTRAINT
CONTAINING	CONTAINS
CONTINUE	CONTROLROW

COPY
CROSS
CURRENT
CURRENT_TIMESTAMP
DATABASE
DATETIME
DEALLOCATE
DECIMAL
DELETE
DESCENDING
DISK
DIV
DOUBLE
DUMP
ENCLOSED
ERROREXIT
EXCEPT
EXECUTE
EXPLAIN
EXTRACT
FIELD
FILLFACTOR
FLOPPY
FOREIGN
FREETEXTTABLE
FUNCTION
GLOBAL
GRANT
HOLDLOCK
IF
INDEX
INNER
INSENSITIVE
INTEGER
INTO
JOIN
LANGUAGE
LEFT
LIKE
LINES
LOCAL
LONG
MATCH
MESSAGE
COUNT
CSTRING
CURRENT_DATE
CURRENT_USER
DATABASES
DAY
DEBUG
DECLARE
DENY
DESCRIBE
DISTINCT
DO
DROP
ELSE
END
ESCAPE
EXCEPTION
EXISTS
EXTEND
FALSE
FIELDS
FILTER
FOR
FOUND
FROM
GENERATOR
GO
GROUP
HOUR
IN
INDICATOR
INOUT
INSERT
INTERSECT
IS
KEY
LAST
LENGTH
LIMIT
LISTEN
LOCK
LOWER
MAX
MIN

MIRROREXIT	MODULE
MONTH	MOVE
NATIONAL	NATURAL
NEXT	NEW
NOCHECK	NONCLUSTERED
NOT	NULL
NUMERIC	OF
OFFSET	OFFSETS
ONCE	ONLY
OPTION	OR
OUTER	OUTPUT
OVERFLOW	OVERLAPS
PAGE	PAGES
PARTIAL	PASSWORD
PERM	PERMANENT
PLAN	POSITION
PREPARE	PRIMARY
PRIOR	PRIVILEGES
PROCEDURE	PROCESSEXIT
PUBLIC	PURGE
READ	READTEXT
REFERENCES	REGEXP
RENAME	REPEAT
REPLICATION	REQUIRE
RESERVING	RESET
RESTRICT	RETAIN
RETURNS	REVOKE
ROLLBACK	ROLLUP
RULE	SAVE
SCHEMA	SECOND
SEGMENT	SELECT
SEPARATOR	SEQUENCE
SET	SETUSER
SHARED	SHOW
SINGULAR	SIZE
SNAPSHOT	SOME
SPACE	SQL
SQLERROR	STABILITY
STARTS	STATISTICS
SUM	SUSPEND
TABLES	TAPE
TEMPORARY	TEXT
THEN	TIME
TO	TOP

TRAN	TRANSACTION
TRIGGER	TRIM
TRUNCATE	UNCOMMITTED
UNIQUE	UNTIL
UPDATETEXT	UPPER
USE	USER
VALUE	VALUES
VARIABLE	VARYING
VIEW	VOLUME
WAITFOR	WHEN
WHILE	WITH
WRITE	WRITETEXT
YEAR	ZONE

常用SQL语句速查

ALTER TABLE

ALTER TABLE用来更新现存表的模式。可以用CREATE TABLE来创建一个新表。详情可参见第17课。

COMMIT

COMMIT用来将事务写入数据库。详情可参见第20课。

CREATE INDEX

CREATE INDEX用来为一列或多列创建索引。详情可参见第22课。

CREATE TABLE

CREATE TABLE用来创建新的数据库表。可以用ALTER TABLE来更新一个现存表的模式。详情可参见第17课。

CREATE VIEW

CREATE VIEW用来创建一个或多个表的视图。详情可参见第18课。

DELETE

DELETE用来从表中删除一行或多行。详情可参见第16课。

DROP

DROP用来永久性地删除数据库对象（表、视图、索引等）。详情可参见第17课和第18课。

INSERT

INSERT用来对表添加一个新行。详情可参见第15课。

INSERT SELECT

INSERT SELECT用来将SELECT的结果插入到表中。详情可参见第15课。

ROLLBACK

ROLLBACK用来撤销事务块。详情可参见第20课。

SELECT

SELECT用来从一个或多个表（或视图）中检索数据。详情可参见第2课、第3课和第4课（第2课到第14课从不同方面涉及了SELECT）。

UPDATE

UPDATE用来对表中的一行或多行进行更新。详情可参见第16课。

版权信息

书名：Spark快速大数据分析

作者：[美] Holden Karau Andy Konwinski,Patrick Wendell [加] Matei Zaharia

译者：王道远

ISBN：978-7-115-40309-4

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

091507240605ToBeReplacedWithUserId

版权声明
O'Reilly Media, Inc. 介绍
业界评论
推荐序
译者序
序
前言
读者对象
本书结构
相关书籍
排版约定
使用代码示例
Safari® Books Online
联系我们
致谢
第 1 章　Spark 数据分析导论
1.1　Spark是什么
1.2　一个大一统的软件栈
1.2.1　Spark Core
1.2.2　Spark SQL
1.2.3　Spark Streaming
1.2.4　MLlib
1.2.5　GraphX
1.2.6　集群管理器
1.3　Spark的用户和用途
1.3.1　数据科学任务
1.3.2　数据处理应用
1.4　Spark简史
1.5　Spark的版本和发布
1.6　Spark的存储层次
第 2 章　Spark 下载与入门
2.1　下载Spark

2.2 Spark中Python和Scala的shell
2.3 Spark核心概念简介
2.4 独立应用
2.4.1 初始化SparkContext
2.4.2 构建独立应用
2.5 总结
第 3 章 RDD 编程
3.1 RDD基础
3.2 创建RDD
3.3 RDD操作
3.3.1 转化操作
3.3.2 行动操作
3.3.3 惰性求值
3.4 向Spark传递函数
3.4.1 Python
3.4.2 Scala
3.4.3 Java
3.5 常见的转化操作和行动操作
3.5.1 基本RDD
3.5.2 在不同RDD类型间转换
3.6 持久化(缓存)
3.7 总结
第 4 章 键值对操作
4.1 动机
4.2 创建Pair RDD
4.3 Pair RDD的转化操作
4.3.1 聚合操作
4.3.2 数据分组
4.3.3 连接
4.3.4 数据排序
4.4 Pair RDD的行动操作
4.5 数据分区（进阶）

4.5.1　获取RDD的分区方式
4.5.2　从分区中获益的操作
4.5.3　影响分区方式的操作
4.5.4　示例：PageRank
4.5.5　自定义分区方式
4.6　总结
第 5 章　数据读取与保存
5.1　动机
5.2　文件格式
5.2.1　文本文件
5.2.2　JSON
5.2.3　逗号分隔值与制表符分隔值
5.2.4　SequenceFile
5.2.5　对象文件
5.2.6　Hadoop输入输出格式
5.2.7　文件压缩
5.3　文件系统
5.3.1　本地/“常规”文件系统
5.3.2　Amazon S3
5.3.3　HDFS
5.4　Spark SQL中的结构化数据
5.4.1　Apache Hive
5.4.2　JSON
5.5　数据库
5.5.1　Java数据库连接
5.5.2 Cassandra
5.5.3　HBase
5.5.4　Elasticsearch
5.6　总结
第 6 章　Spark 编程进阶
6.1　简介
6.2　累加器

6.2.1　累加器与容错性
6.2.2　自定义累加器
6.3　广播变量
广播的优化
6.4　基于分区进行操作
6.5　与外部程序间的管道
6.6　数值RDD的操作
6.7　总结
第 7 章　在集群上运行 Spark
7.1　简介
7.2　Spark运行时架构
7.2.1　驱动器节点
7.2.2　执行器节点
7.2.3　集群管理器
7.2.4　启动一个程序
7.2.5　小结
7.3　使用spark-submit部署应用
7.4　打包代码与依赖
7.4.1　使用Maven构建的用Java编写的Spark应用
7.4.2　使用sbt构建的用Scala编写的Spark应用
7.4.3　依赖冲突
7.5　Spark应用内与应用间调度
7.6　集群管理器
7.6.1　独立集群管理器
7.6.2　Hadoop YARN
7.6.3　Apache Mesos
7.6.4　Amazon EC2
7.7　选择合适的集群管理器
7.8　总结
第 8 章　Spark 调优与调试
8.1　使用SparkConf配置Spark
8.2　Spark执行的组成部分：作业、任务和步骤

8.3　查找信息
8.3.1　Spark网页用户界面
8.3.2　驱动器进程和执行器进程的日志
8.4　关键性能考量
8.4.1　并行度
8.4.2　序列化格式
8.4.3　内存管理
8.4.4　硬件供给
8.5　总结
第 9 章　Spark SQL
9.1　连接Spark SQL
9.2　在应用中使用Spark SQL
9.2.1　初始化Spark SQL
9.2.2　基本查询示例
9.2.3　SchemaRDD
9.2.4　缓存
9.3　读取和存储数据
9.3.1　Apache Hive
9.3.2　Parquet
9.3.3　JSON
9.3.4　基于RDD
9.4　JDBC/ODBC服务器
9.4.1　使用Beeline
9.4.2　长生命周期的表与查询
9.5　用户自定义函数
9.5.1　Spark SQL UDF
9.5.2　Hive UDF
9.6　Spark SQL性能
性能调优选项
9.7　总结
第 10 章　Spark Streaming
10.1　一个简单的例子

10.2　架构与抽象
10.3　转化操作
10.3.1　无状态转化操作
10.3.2　有状态转化操作
10.4　输出操作
10.5　输入源
10.5.1　核心数据源
10.5.2　附加数据源
10.5.3　多数据源与集群规模
10.6　24/7不间断运行
10.6.1　检查点机制
10.6.2　驱动器程序容错
10.6.3　工作节点容错
10.6.4　接收器容错
10.6.5　处理保证
10.7　Streaming用户界面
10.8　性能考量
10.8.1　批次和窗口大小
10.8.2　并行度
10.8.3　垃圾回收和内存使用
10.9　总结
第 11 章基于 MLlib 的机器学习
11.1　概述
11.2　系统要求
11.3　机器学习基础
示例：垃圾邮件分类
11.4　数据类型
操作向量
11.5　算法
11.5.1　特征提取
11.5.2　统计
11.5.3　分类与回归

11.5.4　聚类
11.5.5　协同过滤与推荐
11.5.6　降维
11.5.7　模型评估
11.6　一些提示与性能考量
11.6.1　准备特征
11.6.2　配置算法
11.6.3　缓存RDD以重复使用
11.6.4　识别稀疏程度
11.6.5　并行度
11.7　流水线API
11.8　总结
作者简介
封面介绍

版权声明

© 2015 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2015. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2015。 简体中文版由人民邮电出版社出版，2015。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，

并且切实地按照 **Yogi Berra** 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。'回顾过去，**Tim** 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——***Linux Journal***

推荐序

近年来大数据逐渐升温，经常有人问起大数据为何重要。我们处在一个数据爆炸的时代，大量涌现的智能手机、平板、可穿戴设备及物联网设备每时每刻都在产生新的数据。当今世界，有 90% 的数据是在过去短短两年内产生的。到 2020 年，将有 500 多亿台的互联设备产生 Zeta 字节级的数据。带来革命性改变的并非海量数据本身，而是我们如何利用这些数据。大数据解决方案的强大在于它们可以快速处理大规模、复杂的数据集，可以比传统方法更快、更好地生成洞见。

一套大数据解决方案通常包含多个重要组件，从存储、计算和网络等硬件层，到数据处理引擎，再到利用改良的统计和计算算法、数据可视化来获得商业洞见的分析层。这中间，数据处理引擎起到了十分重要的作用。毫不夸张地说，数据处理引擎之于大数据就像 CPU 之于计算机，或大脑之于人类。

早在 2009 年，Matei Zaharia 在加州大学伯克利分校的 AMPLab 进行博士研究时创立了 Spark 大数据处理和计算框架。不同于传统的数据处理框架，Spark 基于内存的基本类型（primitive）为一些应用程序带来了 100 倍的性能提升。Spark 允许用户程序将数据加载到集群内存中用于反复查询，非常适用于大数据和机器学习，日益成为最广泛采用的大数据模块之一。包括 Cloudera 和 MapR 在内的大数据发行版也在发布时添加了 Spark。

目前，Spark 正在促使 Hadoop 和大数据生态系统发生演变，以更好地支持端到端的大数据分析需求，例如：Spark 已经超越 Spark 核心，发展到了 Spark streaming、SQL、MLlib、GraphX、SparkR 等模块。学习 Spark 和它的各个内部构件不仅有助于改善大数据处理速度，还能帮助开发者和数据科学家更轻松地创建分析应用。从企业、医疗、交通到零售业，Spark 这样的大数据解决方案正以前所未见的力量推进着商业洞见的形成，带来更多更好的洞见以加速决策制定。

在过去几年中，我的部门有机会与本书的作者合作，向 Apache Spark 社区贡献成果，并在英特尔架构上优化各种大数据和 Spark 应用。《Spark 快速大数据分析》的出版为开发者和数据科学家提供了丰富的 Spark 知识。更重要的是，这本书不是简单地教开发者如何使用 Spark，而是更深入介绍了 Spark 的内部构成，并通过各种实例展示了如何优化大数据应用。我向大家推荐这本书，或更具体点，推荐

这本书里提倡的优化方法和思路，相信它们能帮助你创建出更好的大数据应用。

英特尔软件服务事业部全球大数据技术中心总经理 马子雅

2015 年 **7** 月于加州圣克拉拉

Big data is getting hot in recent years. Quite often, folks ask why big data is a big deal. We are in the era of data explosion, with the emergence of smart phones,tablets, wearables, IoT devices, etc. Ninety percent of the data in the world today was generated in just the past two years. By 2020, we will see ＞50B devices connected and Zeta byte data created. It is not the quantity of the data that is revolutionary. It is that we can now do something with it that's revolutionary. The power of big data solutions is they can process large and complex data sets very fast, generate better and faster insights than conventional methods.

A big data solution suite can consist of several critical components, from the hardware layer like storage, compute and network, to data processing engine, to analytics layer where business insights are generated using improved statistical & computational algorithms and data visualization. Among all, the data processing engine is one most critical player. It is not overstating that the data processing engine for big data is like CPU for a computer or brain for a human being.

Spark was initially started for the purpose of creating a big data processing and computing framework, when Matei Zaharia was doing his Ph.D. research at UC Berkeley AMPLab in 2009. Different from the traditional data processing framework, Spark's in-memory primitives provide performance up to 100 times faster for certain applications. By allowing user programs to load data into a cluster's memory and query it repeatedly, Spark is well-suited for big data and machine learning use cases. Spark is becoming one best adopted among all big data modules. Big Data Distributions like Cloudera, MapR now all include Spark into their distributions.

Spark is now evolving the Hadoop and big data ecosystem to better

support the end-to-end big data analytics needs, e.g. Spark grew beyond Spark core to Spark streaming, SQL, MLlib, GraphX, SparkR, etc. Learning Spark and its internals will not just help improve the processing speed for big data, but also help developers and data scientists create analytics applications with more ease. With big data solutions like Spark, we expect to see significant improvement with business insights which will help expedite the decision making —like we've never seen before, from enterprise, healthcare, transportation, and retail.

Over the years, my organization had the opportunities to work with authors of this book, contribute to Apache Spark, and optimize various Big Data and Spark application on Intel Architecture. The publication of Learning Spark offers developers and data scientists'extensive knowledge on Spark. Moreover, Learning Spark does not simply try to tell the developers how to use Spark, it also addresses the internals and shows various examples of how to improve your big data applications. I recommend Learning Spark—that this book, and, more specifically, the method it espouses, will change your big data application for the better.

Ziya Ma, General Manager of the global Big Data Technologies organization,

SSG STO, Intel Corp.

Santa Clara, California, July 2015

译者序

大数据是近几年广受关注的一个概念。今天，互联网不断发展，逐渐深入我们生活的各个层面，随之而来的是数据量的指数级增长。很久以前，人类就学会了通过分析数据获取有价值的结论。有时，影响结论的因素过多，采样的数据无法有效保留所有因素的影响，得出的结论就不够有效。如果不使用采样，而原始数据规模巨大，我们就需要改进数据处理的手段。从人工统计到利用一些传统的计算机软件进行分析，再到 MapReduce 模型，随着数据规模不断增长，我们处理数据的方式也在不断升级。如今，硬件产业的不断发展使得内存计算成为了可能，Spark 由此出现，并且像它的名字一样，以星火之势，迅速赢得了工业界的青睐。

《Spark 快速大数据分析》是一本为 Spark 初学者准备的书，它没有过多深入实现细节，而是更多关注上层用户的具体用法。不过，本书绝不仅仅限于 Spark 的用法，它对 Spark 的核心概念和基本原理也有较为全面的介绍，让读者能够知其然且知其所以然。

Spark 只是一个通用计算框架，利用 Spark 实现的应用才是其真正价值所在。我们很欣慰地看到，国内的许多知名互联网公司已经利用 Spark 创造出了难以估量的价值。本书的读者不妨也尝试把 Spark 应用到实践中，去探寻数据海洋里的无尽瑰宝。

本书得以完成，离不开各方支持。感谢人民邮电出版社图灵公司的李松峰老师、岳新欣老师、张曼老师，他们为本译稿的出版提供了大力支持。感谢本人所在的英特尔亚太研发有限公司大数据团队，其中程浩、孙锐、俞育才、张李晔分别负责了本书各部分的审校工作，黄洁、邵赛赛、史鸣飞也为本书的翻译工作提供了帮助。感谢 Databricks 的连城学长，他促成了我与出版社的合作。在翻译的过程中，来自家人与朋友的理解和支持也让我深深感动。

如本书所述，Spark 是一个大一统的软件栈，涉及方方面面的知识，为本书的翻译增加了不少难度。尽管译者一直努力保证翻译的准确性，由于学识有限，难免会有疏忽之处。而大数据作为一门新兴学科，许多术语尚未有约定俗成的译法。Spark 也在不断发展中，本书英文稿是根据 Spark 1.2 编纂，而译者也尽量标注了直至 Spark 1.4 为止（翻译时的最新版本）引入的一些变化。如果读者发现了本书中的不足或错误之处，恳请批评指正。我的电子邮箱是：

me@daoyuan.wang。

王道远

2015 年夏

序

Spark 作为下一代大数据处理引擎，在非常短的时间里崭露头角，并且以燎原之势席卷业界。Spark 对曾经引爆大数据产业革命的 Hadoop MapReduce 的改进主要体现在这几个方面：首先，Spark 速度更快；其次，Spark 丰富的 API 带来了更强大的易用性；最后，Spark 不单单支持传统批处理应用，更支持交互式查询、流式计算、机器学习、图计算等各种应用，满足各种不同应用场景下的需求。

我很荣幸能够一直密切地参与到 Spark 的开发中，伴随 Spark 一路走来，看着 Spark 从草稿纸上的原型成长为当下最活跃的大数据开源项目。如今，Spark 已经成为 Apache 基金会下最为活跃的项目之一。不仅如此，我也为结识 Spark 项目创始人 Matei Zaharia 以及其他几位 Spark 长期开发者 Patrick Wendell、Andy Konwinski 和 Holden Karau 感到由衷高兴。正是他们四位完成了本书的著作工作。

随着 Spark 的迅速流行，相关优秀参考资料匮乏的问题顿时突显出来。本书共有 11 章，包含许多专为渴望学习 Spark 的数据科学家、学生、开发者们设计的具体实例，大大缓解了 Spark 缺少优秀参考资料的问题。即使是没有大数据方面背景知识的读者，也可以把本书作为入门大数据领域的明智之选。我真挚地希望这本书能引领你和其他读者走进大数据这个令人激动的新领域，在多年之后依然令你回味无穷。

——Databricks 公司首席执行官，加州大学伯克利分校 **AMPlab** 联合主任 **Ion Stoica**

前言

随着并行数据分析变得越来越流行，各行各业的工作者都迫切需要更好的数据分析工具。Spark 应运而生，并且迅速火了起来。作为 MapReduce 的继承者，Spark 主要有三个优点。首先，Spark 非常好用。由于高级 API 剥离了对集群本身的关注，你可以专注于你所要做的计算本身，只需在自己的笔记本电脑上就可以开发 Spark 应用。其次，Spark 很快，支持交互式使用和复杂算法。最后，Spark 是一个通用引擎，可用它来完成各种各样的运算，包括 SQL 查询、文本处理、机器学习等，而在 Spark 出现之前，我们一般需要学习各种各样的引擎来分别处理这些需求。这三大优点也使得 Spark 可以作为学习大数据的一个很好的起点。

本书主要介绍 Spark，让读者能够轻松入门并玩转 Spark。你能从本书中学到如何让 Spark 在你的电脑上运行起来，并且通过交互式操作来学习 Spark 的 API。我们也会讲解一些用 Spark 作数据操作和分布式执行时的细节。最后，本书会带你畅游 Spark 上一些高级的程序库，包括机器学习、流处理、图计算和 SQL 查询。我们希望本书能够让你了解 Spark。不论你只有一台电脑还是有一个庞大的集群，Spark 都能成为令你运筹帷幄的数据分析工具。

读者对象

本书的目标读者是数据科学家和工程师。我们选择这两个群体的原因，在于他们能够利用 Spark 去解决一些可能会遇到但是没有办法解决的问题。Spark 提供了功能丰富的数据操作库（例如 MLlib），可以帮助数据科学家利用他们自己的统计学背景知识，研究数据集大小超过单机所能处理极限的数据问题。与此同时，工程师们则可以从本书中学习和利用 Spark 编写通用的分布式程序并运维这些应用。工程师和数据科学家都不仅能从本书中学到各自需要的具体技能，而且还能够在各自领域中利用 Spark 解决大型分布式问题。

数据科学家关注如何从数据中发现关联以及建立模型。数据科学家通常有着统计学或者数学背景，他们中的大多数也熟悉 Python 语言、R 语言、SQL 等传统数据分析工具。在本书中，我们不仅会讲到 Spark 中一些机器学习和高级数据分析的程序库，也会把一些 Python 或者 SQL 的应用作为 Spark 使用示例进行展示。如果你是一位数据

科学家，我们希望你读完本书之后，能够在获得更快速度和更大数据规模支持的同时，使用早已熟悉的方式来解决问题。

本书的第二类目标读者是软件工程师。对于工程师，不管你擅长的是 Java 还是 Python，抑或是别的编程语言，我们希望这本书能够教会你如何搭建一个 Spark 集群，如何使用 Spark shell，以及如何编写 Spark 应用程序来解决需要并行处理的问题。如果你熟悉 Hadoop，你就已经在如何与 HDFS 进行交互以及如何管理集群的领域中领先了一小步。即使你没有 Hadoop 经验也不用担心，我们会在本书中讲解一些基本的分布式执行的概念。

不论你是数据分析师还是工程师，如果想读透这本书，就应当对 Python、Java、Scala 或者一门类似的编程语言有一些基本了解。另外，我们假设你已经有了关于数据存储的解决方案，所以不会讲到如何搭建一个数据存储系统，不过我们会介绍如何在常见的数据存储系统上读取和保存数据。即使你没用过这些编程语言也不必担心，有很多优秀的学习资源可以帮助你理解这些语言，我们在下文的相关书籍中列举了一些。

本书结构

本书结构清晰，章节是按照从前到后依次阅读的顺序组织的。在每一章的开头，我们会说明本章中的哪些小节对于数据科学家们更重要，而哪些小节则对于工程师们更为有用。话虽如此，我们还是希望书中的所有内容对两类读者都能有一定的帮助。

前两章将会带你入门，让你在自己的电脑上搭好一个基础的 Spark，并且让你对于用 Spark 能做什么有一个基本的概念。等我们弄明白了 Spark 的目标和 Spark 的安装之后，就会着重介绍 Spark shell。Spark shell 是开发 Spark 应用原型时非常有用的工具。后续几章则会详细介绍 Spark API、如何将 Spark 应用运行在集群上，以及 Spark 所提供的更高层的程序库支持，例如 SQL（数据库支持）和 MLlib（机器学习库）。

相关书籍

如果你是一个没有太多 Python 经验的数据科学家，那么我们向你推荐 *Learning Python*（O'Reilly）和 *Head First Python*（O'Reilly）。如果你已经有了一定的 Python 经验，那么 *Dive Into Python*（http://

www.diveintopython.net/）可以进一步加深你对 Python 的理解。

如果你是个工程师，读完本书之后还希望提高自己的数据分析技能，O'Reilly 出版的 *Machine Learning for Hackers* 和 *Doing Data Science* 是不错的参考书。

本书主要为初学者而写，但我们也正在计划为那些渴望全面理解 Spark 内部原理的人写一本更加深入的书。

排版约定

本书使用了下列排版约定。

- 楷体

  表示新术语。

- 等宽字体（`Constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`Constant width bold`**）

  表示应该由用户输入的命令或其他文本。



该图标表示提示或建议。



该图标表示警告或警示。

使用代码示例

本书中所有的示例代码都可以在 GitHub 上找到。你可以从 https://github.com/databricks/learning-spark 中查看和检出这些代码。示例代码包含 Java、Scala 和 Python 的版本。



Java 版本的示例代码兼容 Java 6 以及更高版本。Java 8 引入了支持 lambda 的新语法，可以更方

便地编写内联函数而简化 Spark 代码。由于许多机构还没有开始使用 Java 8，我们决定在我们的大多数示例中不使用这一新语法。如果你对 Java 8 的语法很感兴趣，可以查阅 Databricks 关于 Java 8 语法的博文（http://databricks.com/blog/2014/04/14/spark-with-java-8.html）。有一些示例程序也会移植到 Java 8 上，并且发布在本书的 GitHub 代码仓库中。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Web Development with Node and Express* by Ethan Brown (O'Reilly). Copyright 2014 Ethan Brown, 978-1-491-94930-6."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online



Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal

Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://bit.ly/web_dev_node_express

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

感谢 Joseph Bradley、Dave Bridgeland、Chaz Chandler、Mick Davies、Sam DeHority、Vida Ha、Andrew Gal、Michael Gregson、Jan Joeppen、Stephan Jou、Jeff Martinez、Josh Mahonin、Andrew Or、Mike Patterson、Josh Rosen、Bruce Szalwinski、Xiangrui Meng、Reza Zadeh 等审阅者，他们为本书的写作提出了宝贵的意见。

特别感谢 David Andrzejewski、David Buttler、Juliet Hougland、Marek Kolodziej、Taka Shinagawa、Deborah Siegel、Normen Müller 博士、Ali Ghodsi、Sameer Farooqui 等人，他们为大部分章节提供了详细的反馈，并且帮助指出了许多至关重要的改进之处。

我们还要感谢参与编辑和编写部分章节的主题专家。第 10 章是在我们与 Tathagata Das 的紧密合作下共同完成的。Tathagata 给了我们巨大的帮助，他的工作包括且不限于阐明示例、回答疑问、改进排版以及相关技术的贡献。Michael Armbrust 帮助我们审校了 Spark SQL 相关章节。在第 11 章中，Joseph Bradley 为 MLlib 模块提供了介绍性示例。Reza Zadeh 为关于降维的部分提供了图文描述和代码示例。Xiangrui Meng、Joseph Bradley 和 Reza Zadeh 也为 MLlib 章节提供了编审和关于技术细节的反馈。

第 1 章　Spark 数据分析导论

本章会从宏观角度介绍 Spark 到底是什么。如果你已经对 Spark 和相关组件有一定了解，你可以选择直接从第 2 章开始读。

1.1　Spark是什么

Spark 是一个用来实现快速而通用的集群计算的平台。

在速度方面，Spark 扩展了广泛使用的 MapReduce 计算模型，而且高效地支持更多计算模式，包括交互式查询和流处理。在处理大规模数据集时，速度是非常重要的。速度快就意味着我们可以进行交互式的数据操作，否则我们每次操作就需要等待数分钟甚至数小时。Spark 的一个主要特点就是能够在内存中进行计算，因而更快。不过即使是必须在磁盘上进行的复杂计算，Spark 依然比 MapReduce 更加高效。

总的来说，Spark 适用于各种各样原先需要多种不同的分布式平台的场景，包括批处理、迭代算法、交互式查询、流处理。通过在一个统一的框架下支持这些不同的计算，Spark 使我们可以简单而低耗地把各种处理流程整合在一起。而这样的组合，在实际的数据分析过程中是很有意义的。不仅如此，Spark 的这种特性还大大减轻了原先需要对各种平台分别管理的负担。

Spark 所提供的接口非常丰富。除了提供基于 Python、Java、Scala 和 SQL 的简单易用的 API 以及内建的丰富的程序库以外，Spark 还能和其他大数据工具密切配合使用。例如，Spark 可以运行在 Hadoop 集群上，访问包括 Cassandra 在内的任意 Hadoop 数据源。

1.2　一个大一统的软件栈

Spark 项目包含多个紧密集成的组件。Spark 的核心是一个对由很多计算任务组成的、运行在多个工作机器或者是一个计算集群上的应用进行调度、分发以及监控的计算引擎。由于 Spark 的核心引擎有着速度快和通用的特点，因此 Spark 还支持为各种不同应用场景专门设计的高级组件，比如 SQL 和机器学习等。这些组件关系密切并且可以相互调用，这样你就可以像在平常软件项目中使用程序库那样，组合

使用这些的组件。

各组件间密切结合的设计原理有这样几个优点。首先，软件栈中所有的程序库和高级组件都可以从下层的改进中获益。比如，当 Spark 的核心引擎新引入了一个优化时，SQL 和机器学习程序库也都能自动获得性能提升。其次，运行整个软件栈的代价变小了。不需要运行 5 到 10 套独立的软件系统了，一个机构只需要运行一套软件系统即可。这些代价包括系统的部署、维护、测试、支持等。这也意味着 Spark 软件栈中每增加一个新的组件，使用 Spark 的机构都能马上试用新加入的组件。这就把原先尝试一种新的数据分析系统所需要的下载、部署并学习一个新的软件项目的代价简化成了只需要升级 Spark。

最后，密切结合的原理的一大优点就是，我们能够构建出无缝整合不同处理模型的应用。例如，利用 Spark，你可以在一个应用中实现将数据流中的数据使用机器学习算法进行实时分类。与此同时，数据分析师也可以通过 SQL 实时查询结果数据，比如将数据与非结构化的日志文件进行连接操作。不仅如此，有经验的数据工程师和数据科学家还可以通过 Python shell 来访问这些数据，进行即时分析。其他人也可以通过独立的批处理应用访问这些数据。IT 团队始终只需要维护一套系统即可。

Spark 的各个组件如图 1-1 所示，下面来依次简要介绍它们。

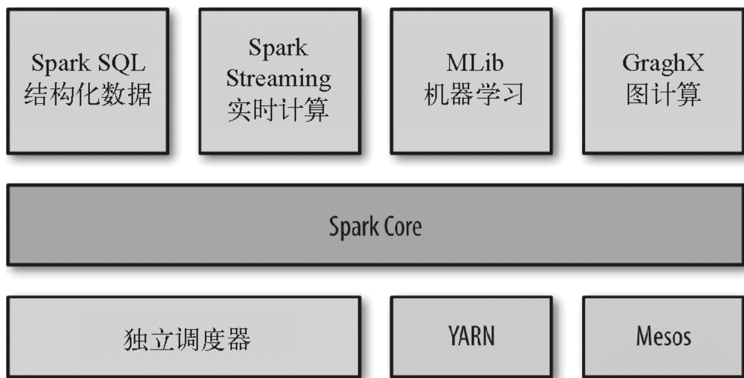


图 **1-1**：**Spark** 软件栈

1.2.1　Spark Core

Spark Core 实现了 Spark 的基本功能，包含任务调度、内存管理、错误恢复、与存储系统交互等模块。Spark Core 中还包含了对弹性分布

式数据集（resilient distributed dataset，简称 RDD）的 API 定义。RDD 表示分布在多个计算节点上可以并行操作的元素集合，是 Spark 主要的编程抽象。Spark Core 提供了创建和操作这些集合的多个 API。

1.2.2 Spark SQL

Spark SQL 是 Spark 用来操作结构化数据的程序包。通过 Spark SQL，我们可以使用 SQL 或者 Apache Hive 版本的 SQL 方言（HQL）来查询数据。Spark SQL 支持多种数据源，比如 Hive 表、Parquet 以及 JSON 等。除了为 Spark 提供了一个 SQL 接口，Spark SQL 还支持开发者将 SQL 和传统的 RDD 编程的数据操作方式相结合，不论是使用 Python、Java 还是 Scala，开发者都可以在单个的应用中同时使用 SQL 和复杂的数据分析。通过与 Spark 所提供的丰富的计算环境进行如此紧密的结合，Spark SQL 得以从其他开源数据仓库工具中脱颖而出。Spark SQL 是在 Spark 1.0 中被引入的。

在 Spark SQL 之前，加州大学伯克利分校曾经尝试修改 Apache Hive 以使其运行在 Spark 上，当时的项目叫作 Shark。现在，由于 Spark SQL 与 Spark 引擎和 API 的结合更紧密，Shark 已经被 Spark SQL 所取代。

1.2.3 Spark Streaming

Spark Streaming 是 Spark 提供的对实时数据进行流式计算的组件。比如生产环境中的网页服务器日志，或是网络服务中用户提交的状态更新组成的消息队列，都是数据流。Spark Streaming 提供了用来操作数据流的 API，并且与 Spark Core 中的 RDD API 高度对应。这样一来，程序员编写应用时的学习门槛就得以降低，不论是操作内存或硬盘中的数据，还是操作实时数据流，程序员都更能应对自如。从底层设计来看，Spark Streaming 支持与 Spark Core 同级别的容错性、吞吐量以及可伸缩性。

1.2.4 MLlib

Spark 中还包含一个提供常见的机器学习（ML）功能的程序库，叫作 MLlib。MLlib 提供了很多种机器学习算法，包括分类、回归、聚类、协同过滤等，还提供了模型评估、数据导入等额外的支持功能。MLlib 还提供了一些更底层的机器学习原语，包括一个通用的梯度下降优化算法。所有这些方法都被设计为可以在集群上轻松伸缩的架

构。

1.2.5　GraphX

GraphX 是用来操作图（比如社交网络的朋友关系图）的程序库，可以进行并行的图计算。与 Spark Streaming 和 Spark SQL 类似，GraphX 也扩展了 Spark 的 RDD API，能用来创建一个顶点和边都包含任意属性的有向图。GraphX 还支持针对图的各种操作（比如进行图分割的 `subgraph` 和操作所有顶点的 `mapVertices`），以及一些常用图算法（比如 PageRank 和三角计数）。

1.2.6　集群管理器

就底层而言，Spark 设计为可以高效地在一个计算节点到数千个计算节点之间伸缩计算。为了实现这样的要求，同时获得最大灵活性，Spark 支持在各种集群管理器（cluster manager）上运行，包括 Hadoop YARN、Apache Mesos，以及 Spark 自带的一个简易调度器，叫作独立调度器。如果要在没有预装任何集群管理器的机器上安装 Spark，那么 Spark 自带的独立调度器可以让你轻松入门；而如果已经有了一个装有 Hadoop YARN 或 Mesos 的集群，通过 Spark 对这些集群管理器的支持，你的应用也同样能运行在这些集群上。第 7 章会详细探讨这些不同的选项以及如何选择合适的集群管理器。

1.3　Spark的用户和用途

Spark 是一个用于集群计算的通用计算框架，因此被用于各种各样的应用程序。在前言中我们提到了本书的两大目标读者人群：数据科学家和工程师。仔细分析这两个群体以及他们使用 Spark 的方式，我们不难发现这两个群体使用 Spark 的典型用例并不一致，不过我们可以把这些用例大致分为两类——数据科学应用和数据处理应用。

当然，这种领域和使用模式的划分是比较模糊的。很多人也兼有数据科学家和工程师的能力，有的时候扮演数据科学家的角色进行研究，然后摇身一变成为工程师，熟练地编写复杂的数据处理程序。不管怎样，分开看这两大群体和相应的用例是很有意义的。

1.3.1　数据科学任务

数据科学是过去几年里出现的新学科，关注的是数据分析领域。尽管没有标准的定义，但我们认为数据科学家（data scientist）就是主要

负责分析数据并建模的人。数据科学家有可能具备 SQL、统计、预测建模（机器学习）等方面的经验，以及一定的使用 Python、Matlab 或 R 语言进行编程的能力。将数据转换为更方便分析和观察的格式，通常被称为数据转换（data wrangling），数据科学家也对这一过程中的必要技术有所了解。

数据科学家使用他们的技能来分析数据，以回答问题或发现一些潜在规律。他们的工作流经常会用到即时分析，所以他们可以使用交互式 shell 替代复杂应用的构建，这样可以在最短时间内得到查询语句和一些简单代码的运行结果。Spark 的速度以及简单的 API 都能在这种场景里大放光彩，而 Spark 内建的程序库的支持也使得很多算法能够即刻使用。

Spark 通过一系列组件支持各种数据科学任务。Spark shell 通过提供 Python 和 Scala 的接口，使我们方便地进行交互式数据分析。Spark SQL 也提供一个独立的 SQL shell，我们可以在这个 shell 中使用 SQL 探索数据，也可以通过标准的 Spark 程序或者 Spark shell 来进行 SQL 查询。机器学习和数据分析则通过 MLlib 程序库提供支持。另外，Spark 还能支持调用 R 或者 Matlab 写成的外部程序。数据科学家在使用 R 或 Pandas 等传统数据分析工具时所能处理的数据集受限于单机，而有了 Spark，就能处理更大数据规模的问题。

在初始的探索阶段之后，数据科学家的工作需要被应用到实际中。具体问题包括扩展应用的功能、提高应用的稳定性，并针对生产环境进行配置，使之成为业务应用的一部分。例如，在数据科学家完成初始的调研之后，我们可能最终会得到一个生产环境中的推荐系统，可以整合在网页应用中，为用户提供产品推荐。一般来说，将数据科学家的工作转化为实际生产中的应用的工作是由另外的工程师或者工程师团队完成的，而不是那些数据科学家。

1.3.2　数据处理应用

Spark 的另一个主要用例是针对工程师的。在这里，我们把工程师定义为使用 Spark 开发生产环境中的数据处理应用的软件开发者。这些开发者一般有基本的软件工程概念，比如封装、接口设计以及面向对象的编程思想，他们通常有计算机专业的背景，并且能使用工程技术来设计和搭建软件系统，以实现业务用例。

对工程师来说，Spark 为开发用于集群并行执行的程序提供了一条捷径。通过封装，Spark 不需要开发者关注如何在分布式系统上编程这样的复杂问题，也无需过多关注网络通信和程序容错性。Spark 已经

为工程师提供了足够的接口来快速实现常见的任务，以及对应用进行监视、审查和性能调优。其 API 模块化的特性（基于传递分布式的对象集）使得利用程序库进行开发以及本地测试大大简化。

Spark 用户之所以选择 Spark 来开发他们的数据处理应用，正是因为 Spark 提供了丰富的功能，容易学习和使用，并且成熟稳定。

1.4 Spark简史

Spark 是由一个强大而活跃的开源社区开发和维护的，社区中的开发者们来自许许多多不同的机构。如果你或者你所在的机构是第一次尝试使用 Spark，也许你会对 Spark 这个项目的历史感兴趣。Spark 是于 2009 年作为一个研究项目在加州大学伯克利分校 RAD 实验室（AMPLab 的前身）诞生。实验室中的一些研究人员曾经用过 Hadoop MapReduce。他们发现 MapReduce 在迭代计算和交互计算的任务上表现得效率低下。因此，Spark 从一开始就是为交互式查询和迭代算法设计的，同时还支持内存式存储和高效的容错机制。

2009 年，关于 Spark 的研究论文在学术会议上发表，同年 Spark 项目正式诞生。其后不久，相比于 MapReduce，Spark 在某些任务上已经获得了 10 ～ 20 倍的性能提升。

Spark 最早的一部分用户来自加州伯克利分校的其他研究小组，其中比较著名的有 Mobile Millennium。作为机器学习领域的研究项目，他们利用 Spark 来监控并预测旧金山湾区的交通拥堵情况。仅仅过了短短的一段时间，许多外部机构也开始使用 Spark。如今，有超过 50 个机构将自己添加到了使用 Spark 的机构列表页面（https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark）。在 Spark 社区如火如荼的社区活动 Spark Meetups（http://www.meetup.com/spark-users/）和 Spark 峰会（http://spark-summit.org/）中，许多机构也向大家积极分享他们特有的 Spark 应用场景。除了加州大学伯克利分校，对 Spark 作出贡献的主要机构还有 Databricks、雅虎以及英特尔。

2011 年，AMPLab 开始基于 Spark 开发更高层的组件，比如 Shark（Spark 上的 Hive）[1] 和 Spark Streaming。这些组件和其他一些组件一起被称为伯克利数据分析工具栈（BDAS，https://amplab.cs.berkeley.edu/software/）。

1Shark 已经被 Spark SQL 所取代。

Spark 最早在 2010 年 3 月开源，并且在 2013 年 6 月交给了 Apache 基金会，现在已经成了 Apache 开源基金会的顶级项目。

1.5 Spark的版本和发布

自其出现以来，Spark 就一直是一个非常活跃的项目，Spark 社区也一直保持着非常繁荣的态势。随着版本号的不断更迭，Spark 的贡献者也与日俱增。Spark 1.0 吸引了 100 多个开源程序员参与开发。尽管项目活跃度在飞速地提升，Spark 社区依然保持着常规的发布新版本的节奏。2014 年 5 月，Spark 1.0 正式发布，而本书则主要关注 Spark 1.1.0 以及后续的版本。不过，大多数概念在老版本的 Spark 中依然适用，而大多数示例也能运行在老版本的 Spark 上。

1.6 Spark的存储层次

Spark 不仅可以将任何 Hadoop 分布式文件系统（HDFS）上的文件读取为分布式数据集，也可以支持其他支持 Hadoop 接口的系统，比如本地文件、亚马逊 S3、Cassandra、Hive、HBase 等。我们需要弄清楚的是，Hadoop 并非 Spark 的必要条件，Spark 支持任何实现了 Hadoop 接口的存储系统。Spark 支持的 Hadoop 输入格式包括文本文件、SequenceFile、Avro、Parquet 等。我们会在第 5 章讨论读取和存储时详细介绍如何与这些数据源进行交互。

第 2 章　Spark 下载与入门

在本章中，我们会下载 Spark 并在本地模式下单机运行它。本章是写给 Spark 的所有初学者的，对数据科学家和工程师来说都值得一读。

Spark 可以通过 Python、Java 或 Scala 来使用 1。要用好本书不需要高超的编程技巧，但是确实需要对其中某种语言的语法有基本的了解。我们会尽可能在示例中给出全部三种语言的代码。

> 1Spark 1.4.0 起添加了 R 语言支持。

Spark 本身是用 Scala 写的，运行在 Java 虚拟机（JVM）上。要在你的电脑或集群上运行 Spark，你要做的准备工作只是安装 Java 6 或者更新的版本。如果你希望使用 Python 接口，你还需要一个 Python 解释器（2.6 以上版本）。Spark 尚不支持 Python 32。

> 2Spark 1.4.0 起支持 Python 3。——译者注

2.1　下载Spark

使用 Spark 的第一步是下载和解压缩。我们先从下载预编译版本的 Spark 开始。访问 http://spark.apache.org/downloads.html，选择包类型为“Pre-built for Hadoop 2.4 and later”（为 Hadoop 2.4 及更新版本预编译的版本），然后选择“Direct Download”直接下载。这样我们就可以得到一个压缩的 TAR 文件，文件名为 spark-1.2.0-bin-hadoop2.4.tgz.



Windows 用户如果把 Spark 安装到带有空格的路径下，可能会遇到一些问题。所以我们需要把 Spark 安装到不带空格的路径下，比如 C:\spark 这样的目录中。

你不需要安装 Hadoop，不过如果你已经有了一个 Hadoop 集群或安装好的 HDFS，请下载对应版本的 Spark。你可以在 http://spark.apache.org/downloads.html 里选择所需要的包类型，这会导致下载得到的文件名略有不同。也可以选择从源代码直接编译。你可以从 GitHub 上下载最新代码，也可以在下载页面上选择包类型

为“Source Code”（源代码）进行下载。

>  大多数类 Unix 系统，包括 OSX 和 Linux，都有一个叫 `tar` 的命令行工具，可以用来解压 TAR 文件。如果你的操作系统没有安装 `tar`，可以尝试搜索网络获取免费的 TAR 解压缩工具。比如，如果你使用的是 Windows，可以试一下 7-Zip.

下载好了 Spark 之后，我们要进行解压缩，然后看一看默认的 Spark 发行版中都有些什么。打开终端，将工作路径转到下载的 Spark 压缩包所在的目录，然后解开压缩包。这样会创建出一个和压缩包同名但是没了 .tgz 后缀的新文件夹。接下来我们就把工作路径转到这个新目录下看看里面都有些什么。上面这些步骤可以用如下命令完成：

```
cd ~
tar -xf spark-1.2.0-bin-hadoop2.4.tgz
cd spark-1.2.0-bin-hadoop2.4
ls
```

在 `tar` 命令所在的那一行中，`x` 标记指定 `tar` 命令执行解压缩操作，`f` 标记则指定压缩包的文件名。`ls` 命令列出了 Spark 目录中的内容。我们先来粗略地看一看 Spark 目录中的一些比较重要的文件及目录的名字和作用。

- README.md

  包含用来入门 Spark 的简单的使用说明。

- bin

  包含可以用来和 Spark 进行各种方式的交互的一系列可执行文件，比如本章稍后会讲到的 Spark shell。

- core、streaming、python……

- 包含Spark项目主要组件的源代码。

- examples

  包含一些可以查看和运行的 Spark 程序，对学习 Spark 的 API 非常有帮助。

不要被 Spark 项目数量庞大的文件和复杂的目录结构吓倒，我们会在本书接下来的部分中讲解它们中的很大一部分。就目前来说，我们还是按部就班，先来试试 Spark 的 Python 和 Scala 版本的 shell。让我们从运行一些 Spark 自带的示例代码开始，然后再编写、编译并运行一个我们自己简易的 Spark 程序。

本章我们所做的一切，Spark 都是在本地模式下运行，也就是非分布式模式，这样我们只需要用到一台机器。Spark 可以运行在许多种模式下，除了本地模式，还支持运行在 Mesos 或 YARN 上，也可以运行在 Spark 发行版自带的独立调度器上。我们会在第 7 章详细讲述各种部署模式。

2.2 Spark中Python和Scala的shell

Spark 带有交互式的 shell，可以作即时数据分析。如果你使用过类似 R、Python、Scala 所提供的 shell，或操作系统的 shell（例如 Bash 或者 Windows 中的命令提示符），你也会对 Spark shell 感到很熟悉。然而和其他 shell 工具不一样的是，在其他 shell 工具中你只能使用单机的硬盘和内存来操作数据，而 Spark shell 可用来与分布式存储在许多机器的内存或者硬盘上的数据进行交互，并且处理过程的分发由 Spark 自动控制完成。

由于 Spark 能够在工作节点上把数据读取到内存中，所以许多分布式计算都可以在几秒钟之内完成，哪怕是那种在十几个节点上处理 TB 级别的数据的计算。这就使得一般需要在 shell 中完成的那些交互式的即时探索性分析变得非常适合 Spark。Spark 提供 Python 以及 Scala 的增强版 shell，支持与集群的连接。



本书中大多数示例代码都包含 Spark 支持的所有语言版本，但是交互式 shell 部分只提供了 Python 和 Scala 版本的示例。shell 对于学习 API 是非常有帮助的，因此我们建议读者在 Python 和 Scala 版本的例子中选择一种进行尝试，即便你是 Java 开发者也是如此，毕竟各种语言的 API 是相似的。

展示 Spark shell 的强大之处最简单的方法就是使用某个语言的 shell 作一些简单的数据分析。我们一起按照 Spark 官方文档中的快速入门指南（http://spark.apache.org/docs/latest/quick-start.html）中的示例来做一遍。

第一步是打开 Spark shell。要打开 Python 版本的 Spark shell，也就是我们所说的 PySpark Shell，进入你的 Spark 目录然后输入：

```
bin/pyspark
```

（在 Windows 中则运行 `bin\pyspark`。）如果要打开 Scala 版本的 shell，输入：

```
bin/spark-shell
```

稍等数秒，shell 提示符就会出现。Shell 启动时，你会看到许多日志信息输出。有的时候，由于提示符之后又输出了日志，我们需要按一下回车键，来得到一个清楚的 shell 提示符。图 2-1 是 PySpark shell 启动时的样子。

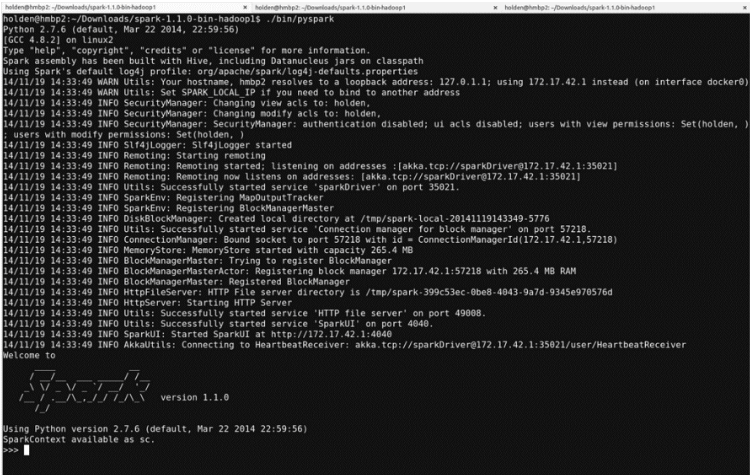


图 **2-1**：默认日志选项下的 **PySpark shell**

如果觉得 shell 中输出的日志信息过多而使人分心，可以调整日志的级别来控制输出的信息量。你需要在 conf 目录下创建一个名为 log4j.properties 的文件来管理日志设置。Spark 开发者们已经在 Spark 中加入了一个日志设置文件的模版，叫作 log4j.properties.template。要让日志看起来不那么啰嗦，可以先把这个日志设置模版文件复制一份到 conf/log4j.properties 来作为日志设置文件，接下来找到下面这一行：

```
```

```
log4j.rootCategory=INFO, console
```

然后通过下面的设定降低日志级别，只显示警告及更严重的信息：

```
log4j.rootCategory=WARN, console
```

这时再打开 shell，你就会看到输出大大减少（图 2-2）。

```
holden@hmbp2:~/Downloads/spark-1.1.0-bin-hadoop1$ ./bin/pyspark
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Spark assembly has been built with Hive, including Datanucleus jars on classpath
14/11/19 14:38:03 WARN Utils: Your hostname, hmbp2 resolves to a loopback address: 127.0.1.1; using 172.17.42.1 instead (on interface docker0)
14/11/19 14:38:03 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
Welcome to
                                   version 1.1.0

Using Python version 2.7.6 (default, Mar 22 2014 22:59:56)
SparkContext available as sc.
>>>
```

图 **2-2**：降低日志级别后的 **PySpark shell**



使用 IPython

IPython 是一个受许多 Python 使用者喜爱的增强版 Python shell，能够提供自动补全等好用的功能。你可以在 http://ipython.org 上找到安装说明。只要把环境变量 IPYTHON 的值设为 1，你就可以使用 `IPython` 了：

```
IPYTHON=1 ./bin/pyspark
```

要使用 IPython Notebook，也就是 Web 版的 IPython，可以运行：

```
IPYTHON_OPTS="notebook" ./bin/pyspark
```

在 Windows 上，像下面这样设置环境变量并运行命令行：

```
set IPYTHON=1
bin\pyspark
```

在 Spark 中，我们通过对分布式数据集的操作来表达我们的计算意图，这些计算会自动地在集群上并行进行。这样的数据集被称为弹性分布式数据集（resilient distributed dataset），简称 RDD。RDD 是 Spark 对分布式数据和计算的基本抽象。

在我们更详细地讨论 RDD 之前，先来使用 shell 从本地文本文件创建一个 RDD 来作一些简单的即时统计。例 2-1 是 Python 版的例子，例 2-2 是 Scala 版的。

例 **2-1**：Python 行数统计

```
>>> lines = sc.textFile("README.md") # 创建一个名为lines的RDD

>>> lines.count() # 统计RDD中的元素个数
127
>>> lines.first() # 这个RDD中的第一个元素，也就是README.md的第一行
u'# Apache Spark'
```

例 **2-2**：Scala 行数统计

```
scala> val lines = sc.textFile("README.md") // 创建一个名为lines的RDD
lines: spark.RDD[String] = MappedRDD[...]

scala> lines.count() // 统计RDD中的元素个数
res0: Long = 127

scala> lines.first() // 这个RDD中的第一个元素，也就是README.md的第一行
res1: String = # Apache Spark
```

要退出任一 shell，按 Ctrl-D。



你可能在日志的输出中注意到了这样一行信息：

INFO SparkUI: Started SparkUI at http://[ipaddress]:4040。你可以由这个地址访问 Spark 用户界面，查看关于任务和集群的各种信息。我们会在第 7 章中详细讨论。

在例 2-1 和例 2-2 中，变量 `lines` 是一个 RDD，是从你电脑上的一个本地的文本文件创建出来的。我们可以在这个 RDD 上运行各种并行操作，比如统计这个数据集中的元素个数在这里就是文本的行数），或者是输出第一个元素。我们会在后续章节中深入探讨 RDD。在此之前，让我们先花些时间来了解 Spark 的基本概念。

2.3　Spark核心概念简介

现在你已经用 shell 运行了你的第一段 Spark 程序，是时候对 Spark 编程作更细致的了解了。

从上层来看，每个 Spark 应用都由一个驱动器程序（driver program）来发起集群上的各种并行操作。驱动器程序包含应用的 `main` 函数，并且定义了集群上的分布式数据集，还对这些分布式数据集应用了相关操作。在前面的例子里，实际的驱动器程序就是 Spark shell 本身，你只需要输入想要运行的操作就可以了。

驱动器程序通过一个 `SparkContext` 对象来访问 Spark。这个对象代表对计算集群的一个连接。shell 启动时已经自动创建了一个 `SparkContext` 对象，是一个叫作 `sc` 的变量。我们可以通过例 2-3 中的方法尝试输出 `sc` 来查看它的类型。

例 **2-3**：查看变量 `sc`

```
>>> sc
<pyspark.context.SparkContext object at 0x1025b8f90>
```

一旦有了 SparkContext，你就可以用它来创建 RDD。在例 2-1 和例 2-2 中，我们调用了 `sc.textFile()` 来创建一个代表文件中各行文本的 RDD。我们可以在这些行上进行各种操作，比如`count()`。

要执行这些操作，驱动器程序一般要管理多个执行器（executor）节点。比如，如果我们在集群上运行 `count()` 操作，那么不同的节点会统计文件的不同部分的行数。由于我们刚才是在本地模式下运行 Spark shell，因此所有的工作会在单个节点上执行，但你可以将这个

shell 连接到集群上来进行并行的数据分析。图 2-3 展示了 Spark 如何在一个集群上运行。

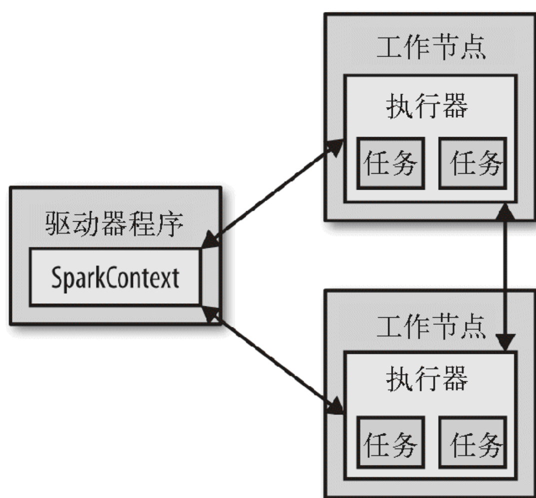


图 **2-3**：**Spark** 分布式执行涉及的组件

最后，我们有很多用来传递函数的 API，可以将对应操作运行在集群上。比如，可以扩展我们的 README 示例，筛选出文件中包含某个特定单词的行。以“Python”这个单词为例，具体代码如例 2-4（Python 版本）和例 2-5（Scala 版本）所示。

例 **2-4**：Python 版本筛选的例子

```
>>> lines = sc.textFile("README.md")

>>> pythonLines = lines.filter(lambda line: "Python" in line)

>>> pythonLines.first()
u'## Interactive Python Shell'
```

例 **2-5**：Scala 版本筛选的例子

```
scala> val lines = sc.textFile("README.md") // 创建一个叫lines的RDD
lines: spark.RDD[String] = MappedRDD[...]

scala> val pythonLines = lines.filter(line => line.contains("Python"))
pythonLines: spark.RDD[String] = FilteredRDD[...]

scala> pythonLines.first()
```

```
res0: String = ## Interactive Python Shell
```

向 Spark 传递函数

如果你对例 2-4 和例 2-5 中的 `lambda` 或者 `=>` 语法不熟悉，可以把它们理解为 Python 和 Scala 中定义内联函数的简写方法。当你在这些语言中使用 Spark 时，你也可以单独定义一个函数，然后把函数名传给 Spark。比如，在 Python 中可以这样做：

```
def hasPython(line):
    return "Python" in line

pythonLines = lines.filter(hasPython)
```

在 Java 中向 Spark 传递函数也是可行的，但是在这种情况下，我们必须把函数定义为实现了 `Function` 接口的类。例如：

```
JavaRDD<String> pythonLines = lines.filter(
  new Function<String, Boolean>() {
    Boolean call(String line) { return line.contains("Python"); }
  }
);
```

Java 8 提供了类似 Python 和 Scala 的 lambda 简写语法。下面就是一个使用这种语法的代码的例子：

```
JavaRDD<String> pythonLines = lines.filter(line -> line.contains(
```

我们会在 3.4 节更深入地讨论如何向 Spark 传递函数。

尽管后面会更详细地讲述 Spark API，我们还是不得不感叹，其实 Spark API 最神奇的地方就在于像 `filter` 这样基于函数的操作也会在集群上并行执行。也就是说，Spark 会自动将函数（比如 `line.contains("Python")`）发到各个执行器节点上。这样，你就可以在单一的驱动器程序中编程，并且让代码自动运行在多个节点上。第 3 章会详细讲述 RDD API。

2.4　独立应用

我们的 Spark 概览中的最后一部分就是如何在独立程序中使用 Spark。除了交互式运行之外，Spark 也可以在 Java、Scala 或 Python 的独立程序中被连接使用。这与在 shell 中使用的主要区别在于你需要自行初始化 SparkContext。接下来，使用的 API 就一样了。

连接 Spark 的过程在各语言中并不一样。在 Java 和 Scala 中，只需要给你的应用添加一个对于 `spark-core` 工件的 Maven 依赖。编写此书时，Spark 的最新版本是 1.2.0，对应的 Maven 索引是：

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.2.0
```

Maven 是一个流行的包管理工具，可以用于任何基于 Java 的语言，让你可以连接公共仓库中的程序库。可以使用 Maven 来构建你的工程，也可以使用其他能够访问 Maven 仓库的工具来进行构建，包括 Scala 的 sbt 工具或者 Gradle 工具。一些常用的集成开发环境（比如 Eclipse）也可以让你直接把 Maven 依赖添加到工程中。

在 Python 中，你可以把应用写成 Python 脚本，但是需要使用 Spark 自带的 `bin/spark-submit` 脚本来运行。`spark-submit` 脚本会帮我们引入 Python 程序的 Spark 依赖。这个脚本为 Spark 的 PythonAPI 配置好了运行环境。你只需要像例 2-6 所示的那样运行脚本即可。

例 **2-6**：运行 Python 脚本

```
bin/spark-submit my_script.py
```

（注意，在 Windows 上需要使用反斜杠来代替斜杠。）

2.4.1　初始化SparkContext

一旦完成了应用与 Spark 的连接，接下来就需要在你的程序中导入 Spark 包并且创建 SparkContext。你可以通过先创建一个 `SparkConf` 对象来配置你的应用，然后基于这个 SparkConf 创建一

个 SparkContext 对象。在例 2-7 至例 2-9 中，我们用各种语言分别示范了这一过程。

例 **2-7**：在 Python 中初始化 Spark

```
from pyspark import SparkConf, SparkContext

conf = SparkConf().setMaster("local").setAppName("My App")
sc = SparkContext(conf = conf)
```

例 **2-8**：在 Scala 中初始化 Spark

```
import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._

val conf = new SparkConf().setMaster("local").setAppName("My App")
val sc = new SparkContext(conf)
```

例 **2-9**：在 Java 中初始化 Spark

```
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaSparkContext;

SparkConf conf = new SparkConf().setMaster("local").setAppName("My App");
JavaSparkContext sc = new JavaSparkContext(conf);
```

这些例子展示了创建 SparkContext 的最基本的方法，你只需传递两个参数：

- **集群 URL**：告诉 Spark 如何连接到集群上。在这几个例子中我们使用的是 `local`，这个特殊值可以让 Spark 运行在单机单线程上而无需连接到集群。
- 应用名：在例子中我们使用的是 `My App`。当连接到一个集群时，这个值可以帮助你在集群管理器的用户界面中找到你的应用。

还有很多附加参数可以用来配置应用的运行方式或添加要发送到集群上的代码。我们会在本书的后续章节中介绍。

在初始化 SparkContext 之后，你可以使用我们前面展示的所有方法

（比如利用文本文件）来创建 RDD 并操控它们。

最后，关闭 Spark 可以调用 SparkContext 的 `stop()` 方法，或者直接退出应用（比如通过 `System.exit(0)` 或者 `sys.exit()`）。

这个快速概览应该已经足够让你在电脑上运行一个独立的 Spark 应用了。如果要了解更高级的配置选项，第 7 章会讲到如何让你的应用连接到一个集群上，包括将你的应用打包，使得代码可以自动发送到工作节点上。就目前而言，参考 Spark 官方文档的快速入门指南（http://spark.apache.org/docs/latest/quick-start.html）就足够了。

2.4.2　构建独立应用

作为一本讲大数据的书，如果没有一个单词数统计的例子，就不能成其为完整的一章。在单机上实现单词数统计很容易，但在分布式框架下，由于要在许多工作节点上读入并组合数据，单词数统计就成了一个很常用的例子。下面我们来学习用 sbt 以及 Maven 来构建并打包一个简单的单词数统计的例程。我们可以把所有的例程构建在一起，但是为了展示最简单的构建过程，我们只保留了最基本的依赖。在 learning-spark-examples/mini-complete-example 目录下，你可以找到这样一个独立的小工程。Java 版本（例 2-10）和 Scala 版本（例 2-11）的例子分别如下所示。

例 **2-10**：Java 版本的单词数统计应用（暂时不需要深究细节）

```
// 创建一个Java版本的Spark Context
SparkConf conf = new SparkConf().setAppName("wordCount");
JavaSparkContext sc = new JavaSparkContext(conf);
// 读取我们的输入数据
JavaRDD<String> input = sc.textFile(inputFile);
// 切分为单词
JavaRDD<String> words = input.flatMap(
  new FlatMapFunction<String, String>() {
    public Iterable<String> call(String x) {
      return Arrays.asList(x.split(" "));
    }});
// 转换为键值对并计数
JavaPairRDD<String, Integer> counts = words.mapToPair(
  new PairFunction<String, String, Integer>(){
    public Tuple2<String, Integer> call(String x){
      return new Tuple2(x, 1);
    }}).reduceByKey(new Function2<Integer, Integer, Integer>(){
      public Integer call(Integer x, Integer y){ return x + y;}});
```

```
// 将统计出来的单词总数存入一个文本文件，引发求值
counts.saveAsTextFile(outputFile);
```

例 2-11：Scala 版本的单词数统计应用（暂时不需要深究细节）

```
// 创建一个Scala版本的Spark Context
val conf = new SparkConf().setAppName("wordCount")
val sc = new SparkContext(conf)
// 读取我们的输入数据
val input = sc.textFile(inputFile)
// 把它切分成一个个单词
val words = input.flatMap(line => line.split(" "))
// 转换为键值对并计数
val counts = words.map(word => (word, 1)).reduceByKey{case (x, y) => x + y
// 将统计出来的单词总数存入一个文本文件，引发求值
counts.saveAsTextFile(outputFile)
```

我们可以使用非常简单的 sbt（例 2-12）或 Maven（例 2-13）构建文件来构建这些应用。由于 Spark Core 包已经在各个工作节点的 classpath 中了，所以我们把对 Spark Core 的依赖标记为 `provided`，这样当我们稍后使用 assembly 方式打包应用时，就不会把 `spark-core` 包也打包到 assembly 包中。

例 2-12：sbt 构建文件

```
name := "learning-spark-mini-example"

version := "0.0.1"

scalaVersion := "2.10.4"

// 附加程序库
libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core" % "1.2.0" % "provided"
)
```

例 2-13：Maven 构建文件

```
<project>
  <groupId>com.oreilly.learningsparkexamples.mini</groupId>
  <artifactId>learning-spark-mini-example</artifactId>
  <modelVersion>4.0.0</modelVersion>
  <name>example</name>
  <packaging>jar</packaging>
```

```
  <version>0.0.1</version>
  <dependencies>
    <dependency> <!-- Spark依赖 -->
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.2.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  <properties>
    <java.version>1.6</java.version>
  </properties>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>   <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <version>3.1</version>
          <configuration>
            <source>${java.version}</source>
            <target>${java.version}</target>
          </configuration> </plugin> </plugin>
      </plugins>
    </pluginManagement>
  </build>
</project>
```



`spark-core` 包被标记为了 `provided`，这是为了控制我们以 assembly 方式打包应用时的行为。第 7 章中会详细讨论这个细节。

一旦敲定了构建方式，我们就可以轻松打包并且使用 `bin/spark-submit` 脚本执行我们的应用了。`spark-submit` 脚本可以为我们配置 Spark 所要用到的一系列环境变量。在 mini-complete-example 目录中，我们可以使用 Scala（例 2-14）或者 Java（例 2-15）进行构建。

例 **2-14**：Scala 构建与运行

```
sbt clean package
$SPARK_HOME/bin/spark-submit \
  --class com.oreilly.learningsparkexamples.mini.scala.WordCount \
  ./target/...  (as above) \
  ./README.md ./wordcounts
```

例 **2-15**：Maven 构建与运行

```
mvn clean && mvn compile && mvn package
$SPARK_HOME/bin/spark-submit \
  --class com.oreilly.learningsparkexamples.mini.java.WordCount \
  ./target/learning-spark-mini-example-0.0.1.jar \
  ./README.md ./wordcounts
```

要了解关于连接应用程序到 Spark 的更多例子，请参考 Spark 官方文档中的快速入门指南（http://spark.apache.org/docs/latest/quick-start.html）一节。第 7 章中也会更详细地讲解如何打包 Spark 应用。

2.5　总结

在本章中，我们讲到了下载并在单机的本地模式下运行 Spark，以及 Spark 的使用方式，包括交互式方式和通过一个独立应用进行调用。另外我们还简单介绍了 Spark 编程的核心概念：通过一个驱动器程序创建一个 SparkContext 和一系列 RDD，然后进行并行操作。在下一章中，我们将会更加深入地介绍如何操作 RDD。

第 3 章　RDD 编程

本章介绍 Spark 对数据的核心抽象——弹性分布式数据集（Resilient Distributed Dataset，简称 RDD）。RDD 其实就是分布式的元素集合。在 Spark 中，对数据的所有操作不外乎创建 RDD、转化已有 RDD 以及调用 RDD 操作进行求值。而在这一切背后，Spark 会自动将 RDD 中的数据分发到集群上，并将操作并行化执行。

由于 RDD 是 Spark 的核心概念，因此数据科学家和工程师都应该读一读本章。我们强烈建议读者在交互式 shell（参见 2.2 节）中亲身尝试一些示例。此外，本章中的示例代码都可以在本书的 GitHub 仓库（https://github.com/databricks/learning-spark）中找到。

3.1　RDD基础

Spark 中的 RDD 就是一个不可变的分布式对象集合。每个 RDD 都被分为多个分区，这些分区运行在集群中的不同节点上。RDD 可以包含 Python、Java、Scala 中任意类型的对象，甚至可以包含用户自定义的对象。

用户可以使用两种方法创建 RDD：读取一个外部数据集，或在驱动器程序里分发驱动器程序中的对象集合（比如 list 和 set）。我们在本书前面的章节中已经见过使用 `SparkContext.textFile()` 来读取文本文件作为一个字符串 RDD 的示例，如例 3-1 所示。

例 **3-1**：在 Python 中使用 `textFile()` 创建一个字符串的 RDD

```
>>> lines = sc.textFile("README.md")
```

创建出来后，RDD 支持两种类型的操作：转化操作（transformation）和行动操作（action）。转化操作会由一个 RDD 生成一个新的 RDD。例如，根据谓词匹配情况筛选数据就是一个常见的转化操作。在我们的文本文件示例中，我们可以用筛选来生成一个只存储包含单词 Python 的字符串的新的 RDD，如例 3-2 所示。

例 **3-2**：调用转化操作 `filter()`

```
>>> pythonLines = lines.filter(lambda line: "Python" in line)
```

另一方面，行动操作会对 RDD 计算出一个结果，并把结果返回到驱动器程序中，或把结果存储到外部存储系统（如 HDFS）中。first() 就是我们之前调用的一个行动操作，它会返回 RDD 的第一个元素，如例 3-3 所示。

例 **3-3**：调用 first() 行动操作

```
>>> pythonLines.first()
u'## Interactive Python Shell'
```

转化操作和行动操作的区别在于 Spark 计算 RDD 的方式不同。虽然你可以在任何时候定义新的 RDD，但 Spark 只会惰性计算这些 RDD。它们只有第一次在一个行动操作中用到时，才会真正计算。这种策略刚开始看起来可能会显得有些奇怪，不过在大数据领域是很有道理的。比如，看看例 3-2 和例 3-3，我们以一个文本文件定义了数据，然后把其中包含 Python 的行筛选出来。如果 Spark 在我们运行 lines = sc.textFile(...) 时就把文件中所有的行都读取并存储起来，就会消耗很多存储空间，而我们马上就要筛选掉其中的很多数据。相反，一旦 Spark 了解了完整的转化操作链之后，它就可以只计算求结果时真正需要的数据。事实上，在行动操作 first() 中，Spark 只需要扫描文件直到找到第一个匹配的行为止，而不需要读取整个文件。

最后，默认情况下，Spark 的 RDD 会在你每次对它们进行行动操作时重新计算。如果想在多个行动操作中重用同一个 RDD，可以使用 RDD.persist() 让 Spark 把这个 RDD 缓存下来。我们可以让 Spark 把数据持久化到许多不同的地方，可用的选项会在表 3-6 中列出。在第一次对持久化的 RDD 计算之后，Spark 会把 RDD 的内容保存到内存中（以分区方式存储到集群中的各机器上），这样在之后的行动操作中，就可以重用这些数据了。我们也可以把 RDD 缓存到磁盘上而不是内存中。默认不进行持久化可能也显得有些奇怪，不过这对于大规模数据集是很有意义的：如果不会重用该 RDD，我们就没有必要浪费存储空间，Spark 可以直接遍历一遍数据然后计算出结果。**1**

1在任何时候都能进行重算是我们为什么把 RDD 描述为“弹性”的原因。当保存 RDD 数据的一台机器失败时，Spark 还可以使用这种特性来重算出丢掉的分区，这一过程对用户是完全透明的。

在实际操作中，你会经常用 `persist()` 来把数据的一部分读取到内存中，并反复查询这部分数据。例如，如果我们想多次对 README 文件中包含 Python 的行进行计算，就可以写出如例 3-4 所示的脚本。

例 **3-4**：把 RDD 持久化到内存中

```
>>> pythonLines.persist

>>> pythonLines.count()
2

>>> pythonLines.first()
u'## Interactive Python Shell'
```

总的来说，每个 Spark 程序或 shell 会话都按如下方式工作。

(1) 从外部数据创建出输入 RDD。

(2) 使用诸如 `filter()` 这样的转化操作对 RDD 进行转化，以定义新的 RDD。

(3) 告诉 Spark 对需要被重用的中间结果 RDD 执行 `persist()` 操作。

(4) 使用行动操作（例如 `count()` 和 `first()` 等）来触发一次并行计算，Spark 会对计算进行优化后再执行。



`cache()` 与使用默认存储级别调用 `persist()` 是一样的。

接下来我们会对这几个步骤逐一详解，并介绍 Spark 中常见的一些 RDD 操作。

3.2　创建RDD

Spark 提供了两种创建 RDD 的方式：读取外部数据集，以及在驱动器程序中对一个集合进行并行化。

创建 RDD 最简单的方式就是把程序中一个已有的集合传给 SparkContext 的 `parallelize()` 方法，如例 3-5 至例 3-7 所示。

这种方式在学习 Spark 时非常有用，它让你可以在 shell 中快速创建出自己的 RDD，然后对这些 RDD 进行操作。不过，需要注意的是，除了开发原型和测试时，这种方式用得并不多，毕竟这种方式需要把你的整个数据集先放在一台机器的内存中。

例 **3-5**：Python 中的 `parallelize()` 方法

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

例 **3-6**：Scala 中的 `parallelize()` 方法

```
val lines = sc.parallelize(List("pandas", "i like pandas"))
```

例 **3-7**：Java 中的 `parallelize()` 方法

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("pandas", "i like par
```

更常用的方式是从外部存储中读取数据来创建 RDD。外部数据集的读取会在第 5 章详细介绍。不过，我们已经接触了用来将文本文件读入为一个存储字符串的 RDD 的方法 `SparkContext.textFile()`，用法如例 3-8 至例 3-10 所示。

例 **3-8**：Python 中的 `textFile()` 方法

```
lines = sc.textFile("/path/to/README.md")
```

例 **3-9**：Scala 中的 `textFile()` 方法

```
val lines = sc.textFile("/path/to/README.md")
```

例 **3-10**：Java 中的 `textFile()` 方法

```
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

3.3　RDD操作

我们已经讨论过，RDD 支持两种操作：转化操作和行动操作。RDD 的转化操作是返回一个新的 RDD 的操作，比如 `map()` 和 `filter()`，而行动操作则是向驱动器程序返回结果或把结果写入外部系统的操作，会触发实际的计算，比如 `count()` 和 `first()`。Spark 对待转化操作和行动操作的方式很不一样，因此理解你正在进行的操作的类型是很重要的。如果对于一个特定的函数是属于转化操作还是行动操作感到困惑，你可以看看它的返回值类型：转化操作返回的是 RDD，而行动操作返回的是其他的数据类型。

3.3.1 转化操作

RDD 的转化操作是返回新 RDD 的操作。我们会在 3.3.3 节讲到，转化出来的 RDD 是惰性求值的，只有在行动操作中用到这些 RDD 时才会被计算。许多转化操作都是针对各个元素的，也就是说，这些转化操作每次只会操作 RDD 中的一个元素。不过并不是所有的转化操作都是这样的。

举个例子，假定我们有一个日志文件 log.txt，内含有若干消息，希望选出其中的错误消息。我们可以使用前面说过的转化操作 `filter()`。不过这一次，我们会展示如何用 Spark 支持的三种语言的 API 分别实现（见例 3-11 至例 3-13）。

例 **3-11**：用 Python 实现 `filter()` 转化操作

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

例 **3-12**：用 Scala 实现 `filter()` 转化操作

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
```

例 **3-13**：用 Java 实现 `filter()` 转化操作

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
  new Function<String, Boolean>() {
    public Boolean call(String x) { return x.contains("error"); }
  }
});
```

注意，filter() 操作不会改变已有的 inputRDD 中的数据。实际上，该操作会返回一个全新的 RDD。inputRDD 在后面的程序中还可以继续使用，比如我们还可以从中搜索别的单词。事实上，要再从 inputRDD 中找出所有包含单词 warning 的行。接下来，我们使用另一个转化操作 union() 来打印出包含 error 或 warning 的行数。下例中用 Python 作了示例，不过 union() 函数的用法在所有三种语言中是一样的。

例 **3-14**：用 Python 进行 union() 转化操作

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD = inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

union() 与 filter() 的不同点在于它操作两个 RDD 而不是一个。转化操作可以操作任意数量的输入 RDD。


要获得与例 3-14 中等价的结果，更好的方法是直接筛选出要么包含 error 要么包含 warning 的行，这样只对 inputRDD 进行一次筛选即可。

最后要说的是，通过转化操作，你从已有的 RDD 中派生出新的 RDD，Spark 会使用谱系图（lineage graph）来记录这些不同 RDD 之间的依赖关系。Spark 需要用这些信息来按需计算每个 RDD，也可以依靠谱系图在持久化的 RDD 丢失部分数据时恢复所丢失的数据。图 3-1 展示了例 3-14 中的谱系图。

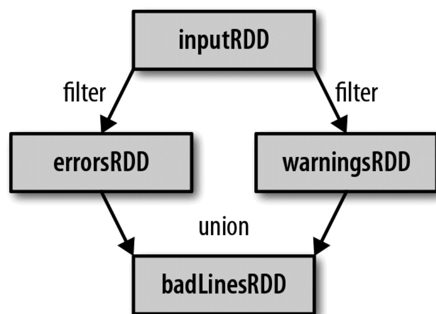


图 **3-1**：日志分析过程中创建出的 **RDD** 谱系图

3.3.2　行动操作

我们已经看到了如何通过转化操作从已有的 RDD 创建出新的 RDD，不过有时，我们希望对数据集进行实际的计算。行动操作是第二种类型的 RDD 操作，它们会把最终求得的结果返回到驱动器程序，或者写入外部存储系统中。由于行动操作需要生成实际的输出，它们会强制执行那些求值必须用到的 RDD 的转化操作。

继续我们在前几章中用到的日志的例子，我们可能想输出关于 `badLinesRDD` 的一些信息。为此，需要使用两个行动操作来实现：用 `count()` 来返回计数结果，用 `take()` 来收集 RDD 中的一些元素，如例 3-15 至例 3-17 所示。

例 **3-15**：在 Python 中使用行动操作对错误进行计数

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
    print line
```

例 **3-16**：在 Scala 中使用行动操作对错误进行计数

```
println("Input had " + badLinesRDD.count() + " concerning lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

例 **3-17**：在 Java 中使用行动操作对错误进行计数

```
System.out.println("Input had " + badLinesRDD.count() + " concerning lines
System.out.println("Here are 10 examples:")
for (String line: badLinesRDD.take(10)) {
  System.out.println(line);
}
```

在这个例子中，我们在驱动器程序中使用 `take()` 获取了 RDD 中的少量元素。然后在本地遍历这些元素，并在驱动器端打印出来。RDD 还有一个 `collect()` 函数，可以用来获取整个 RDD 中的数据。如果你的程序把 RDD 筛选到一个很小的规模，并且你想在本地处理这些数据时，就可以使用它。记住，只有当你的整个数据集能在单台机器的内存中放得下时，才能使用 `collect()`，因此，`collect()` 不能用在大规模数据集上。

在大多数情况下，RDD 不能通过 collect() 收集到驱动器进程中，因为它们一般都很大。此时，我们通常要把数据写到诸如 HDFS 或 Amazon S3 这样的分布式的存储系统中。你可以使用 saveAsTextFile()、saveAsSequenceFile()，或者任意的其他行动操作来把 RDD 的数据内容以各种自带的格式保存起来。我们会在第 5 章讲解导出数据的各种选项。

需要注意的是，每当我们调用一个新的行动操作时，整个 RDD 都会从头开始计算。要避免这种低效的行为，用户可以将中间结果持久化，这会在 3.6 节中介绍。

3.3.3　惰性求值

前面提过，RDD 的转化操作都是惰性求值的。这意味着在被调用行动操作之前 Spark 不会开始计算。这对新用户来说可能与直觉有些相违背之处，但是对于那些使用过诸如 Haskell 等函数式语言或者类似 LINQ 这样的数据处理框架的人来说，会有些似曾相识。

惰性求值意味着当我们对 RDD 调用转化操作（例如调用 map()）时，操作不会立即执行。相反，Spark 会在内部记录下所要求执行的操作的相关信息。我们不应该把 RDD 看作存放着特定数据的数据集，而最好把每个 RDD 当作我们通过转化操作构建出来的、记录如何计算数据的指令列表。把数据读取到 RDD 的操作也同样是惰性的。因此，当我们调用 sc.textFile() 时，数据并没有读取进来，而是在必要时才会读取。和转化操作一样的是，读取数据的操作也有可能会多次执行。


虽然转化操作是惰性求值的，但还是可以随时通过运行一个行动操作来强制 Spark 执行 RDD 的转化操作，比如使用 count()。这是一种对你所写的程序进行部分测试的简单方法。

Spark 使用惰性求值，这样就可以把一些操作合并到一起来减少计算数据的步骤。在类似 Hadoop MapReduce 的系统中，开发者常常花费大量时间考虑如何把操作组合到一起，以减少 MapReduce 的周期数。而在 Spark 中，写出一个非常复杂的映射并不见得能比使用很多简单的连续操作获得好很多的性能。因此，用户可以用更小的操作来组织他们的程序， 这样也使这些操作更容易管理。

3.4　向Spark传递函数

Spark 的大部分转化操作和一部分行动操作，都需要依赖用户传递的函数来计算。在我们支持的三种主要语言中，向 Spark 传递函数的方式略有区别。

3.4.1　Python

在 Python 中，我们有三种方式来把函数传递给 Spark。传递比较短的函数时，可以使用 lambda 表达式来传递，如例 3-2 和例 3-18 所示。除了 lambda 表达式，我们也可以传递顶层函数或是定义的局部函数。

例 **3-18**：在 Python 中传递函数

```
word = rdd.filter(lambda s: "error" in s)

def containsError(s):
    return "error" in s
word = rdd.filter(containsError)
```

传递函数时需要小心的一点是，Python 会在你不经意间把函数所在的对象也序列化传出去。当你传递的对象是某个对象的成员，或者包含了对某个对象中一个字段的引用时（例如 `self.field`），Spark 就会把整个对象发到工作节点上，这可能比你想传递的东西大得多（见例 3-19）。有时，如果传递的类里面包含 Python 不知道如何序列化传输的对象，也会导致你的程序失败。

例 **3-19**：传递一个带字段引用的函数（别这么做！）

```
class SearchFunctions(object):
  def __init__(self, query):
      self.query = query
  def isMatch(self, s):
      return self.query in s
  def getMatchesFunctionReference(self, rdd):
      # 问题：在"self.isMatch"中引用了整个self
      return rdd.filter(self.isMatch)
  def getMatchesMemberReference(self, rdd):
      # 问题：在"self.query"中引用了整个self
      return rdd.filter(lambda x: self.query in x)
```

替代的方案是，只把你所需要的字段从对象中拿出来放到一个局部变

量中，然后传递这个局部变量，如例 3-20 所示。

例 **3-20**：传递不带字段引用的 Python 函数

```
class WordFunctions(object):
  ...
  def getMatchesNoReference(self, rdd):
      # 安全：只把需要的字段提取到局部变量中
      query = self.query
      return rdd.filter(lambda x: query in x)
```

3.4.2 Scala

在 Scala 中，我们可以把定义的内联函数、方法的引用或静态方法传递给 Spark，就像 Scala 的其他函数式 API 一样。我们还要考虑其他一些细节，比如所传递的函数及其引用的数据需要是可序列化的（实现了 Java 的 Serializable 接口）。除此以外，与 Python 类似，传递一个对象的方法或者字段时，会包含对整个对象的引用。这在 Scala 中不是那么明显，毕竟我们不会像 Python 那样必须用 `self` 写出那些引用。类似在例 3-20 中对 Python 执行的操作，我们可以把需要的字段放到一个局部变量中，来避免传递包含该字段的整个对象，如例 3-21 所示。

例 **3-21**：Scala 中的函数传递

```
class SearchFunctions(val query: String) {
  def isMatch(s: String): Boolean = {
    s.contains(query)
  }
  def getMatchesFunctionReference(rdd: RDD[String]): RDD[String] = {
    // 问题："isMatch"表示"this.isMatch"，因此我们要传递整个"this"
    rdd.map(isMatch)
  }
  def getMatchesFieldReference(rdd: RDD[String]): RDD[String] = {
    // 问题："query"表示"this.query"，因此我们要传递整个"this"
    rdd.map(x => x.split(query))
  }
  def getMatchesNoReference(rdd: RDD[String]): RDD[String] = {
    // 安全：只把我们需要的字段拿出来放入局部变量中
    val query_ = this.query
    rdd.map(x => x.split(query_))
  }
}
```

如果在 Scala 中出现了 `NotSerializableException`，通常问题就在于我们传递了一个不可序列化的类中的函数或字段。记住，传递局部可序列化变量或顶级对象中的函数始终是安全的。

3.4.3　Java

在 Java 中，函数需要作为实现了 Spark 的 `org.apache.spark.api.java.function` 包中的任一函数接口的对象来传递。根据不同的返回类型，我们定义了一些不同的接口。我们把最基本的一些函数接口列在表 3-1 中，同时介绍了一些其他的函数接口，在需要返回特殊类型（比如键值对）的数据时使用，参见 3.5.2 节中的“Java”一节。

表**3-1**：标准**Java**函数接口

函数名	实现的方法	用途
`Function<T, R>`	[illegible]	接收一个输入值并返回一个输出值，用于类似 `map()` 和 `filter()` 等操作中
`Function2<T1, T2, R>`	[illegible]	接收两个输入值并返回一个输出值，用于类似 `aggregate()` 和 `fold()` 等操作中
`FlatMapFunction<T, R>`	[illegible]	接收一个输入值并返回任意个输出，用于类似 `flatMap()` 这样的操作中

可以把我们的函数类内联定义为使用匿名内部类（例 3-22），也可以创建一个具名类（例 3-23）。

例 **3-22**：在 Java 中使用匿名内部类进行函数传递

```
RDD<String> errors = lines.filter(new Function<String, Boolean>() {
  public Boolean call(String x) { return x.contains("error"); }
});
```

例 **3-23**：在 Java 中使用具名类进行函数传递

```
class ContainsError implements Function<String, Boolean>() {
  public Boolean call(String x) { return x.contains("error"); }
}

RDD<String> errors = lines.filter(new ContainsError());
```

具体风格的选择取决于个人偏好。不过我们发现顶级具名类通常在组织大型程序时显得比较清晰。使用顶级函数的另一个好处在于你可以给它们的构造函数添加参数，如例 3-24 所示。

例 **3-24**：带参数的 Java 函数类

```
class Contains implements Function<String, Boolean>() {
  private String query;
  public Contains(String query) { this.query = query; }
  public Boolean call(String x) { return x.contains(query); }
}

RDD<String> errors = lines.filter(new Contains("error"));
```

在 Java 8 中，你也可以使用 lambda 表达式来简洁地实现函数接口。由于在本书写作时，Java 8 仍然相对比较新，我们的示例就使用了之前版本的 Java，以更啰嗦的语法来定义函数类。不过，如果使用 lambda 表达式，我们的搜索示例就会变得如例 3-25 所示那样。

例 **3-25**：在 Java 中使用 Java 8 地 lambda 表达式进行函数传递

```
RDD<String> errors = lines.filter(s -> s.contains("error"));
```

如果你对使用 Java 8 的 lambda 表达式感兴趣，请参考 Oracle 的相关文档（http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html）以及 Databricks 关于如何在 Spark 中 使用 lambda 表达式的博客（http://databricks.com/blog/2014/04/14/spark-with-java-8.html）。



匿名内部类和 lambda 表达式都可以引用方法中封装的任意 `final` 变量，因此你可以像在 Python 和 Scala 中一样把这些变量传递给 Spark。

3.5 常见的转化操作和行动操作

本节我们会接触 Spark 中大部分常见的转化操作和行动操作。包含特定数据类型的 RDD 还支持一些附加操作，例如，数字类型的 RDD 支持统计型函数操作，而键值对形式的 RDD 则支持诸如根据键聚合数据的键值对操作。我们也会在后面几节中讲到如何转换 RDD 类型，以及各类型对应的特殊操作。

3.5.1 基本RDD

首先来讲讲哪些转化操作和行动操作受任意数据类型的 RDD 支持。

1. 针对各个元素的转化操作

你很可能会用到的两个最常用的转化操作是 `map()` 和 `filter()`（见图 3-2）。转化操作 `map()` 接收一个函数，把这个函数用于 RDD 中的每个元素，将函数的返回结果作为结果 RDD 中对应元素的值。而转化操作 `filter()` 则接收一个函数，并将 RDD 中满足该函数的元素放入新的 RDD 中返回。

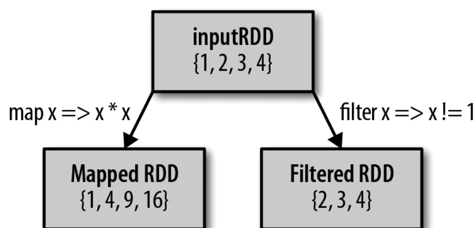


图 **3-2**：从输入 **RDD** 映射与筛选得到的 **RDD**

我们可以使用 `map()` 来做各种各样的事情：可以把我们的 URL 集合中的每个 URL 对应的主机名提取出来，也可以简单到只对各个数字求平方值。`map()` 的返回值类型不需要和输入类型一样。这样如果有一个字符串 RDD，并且我们的 `map()` 函数是用来把字符串解析并返回一个 `Double` 值的，那么此时我们的输入 RDD 类型就是 `RDD[String]`，而输出类型是 `RDD[Double]`。

让我们看一个简单的例子，用 `map()` 对 RDD 中的所有数求平方（如例 3-26 至例 3-28 所示）。

例 **3-26**：Python 版计算 RDD 中各值的平方

```
nums = sc.parallelize([1, 2, 3, 4])
squared = nums.map(lambda x: x * x).collect()
for num in squared:
    print "%i " % (num)
```

例 **3-27**：Scala 版计算 RDD 中各值的平方

```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
println(result.collect().mkString(","))
```

例 3-28：Java 版计算 RDD 中各值的平方

```
JavaRDD<Integer> rdd = sc.parallelize(Arrays.asList(1, 2, 3, 4));
JavaRDD<Integer> result = rdd.map(new Function<Integer, Integer>() {
  public Integer call(Integer x) { return x*x; }
});
System.out.println(StringUtils.join(result.collect(), ","));
```

有时候，我们希望对每个输入元素生成多个输出元素。实现该功能的操作叫作 `flatMap()`。和 `map()` 类似，我们提供给 `flatMap()` 的函数被分别应用到了输入 RDD 的每个元素上。不过返回的不是一个元素，而是一个返回值序列的迭代器。输出的 RDD 倒不是由迭代器组成的。我们得到的是一个包含各个迭代器可访问的所有元素的 RDD。`flatMap()` 的一个简单用途是把输入的字符串切分为单词，如例 3-29 至例 3-31 所示。

例 3-29：Python 中的 `flatMap()` 将行数据切分为单词

```
lines = sc.parallelize(["hello world", "hi"])
words = lines.flatMap(lambda line: line.split(" "))
words.first() # 返回"hello"
```

例 3-30：Scala 中的 `flatMap()` 将行数据切分为单词

```
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
words.first() // 返回"hello"
```

例 3-31：Java 中的 `flatMap()` 将行数据切分为单词

```
JavaRDD<String> lines = sc.parallelize(Arrays.asList("hello world", "hi"));
JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
  public Iterable<String> call(String line) {
    return Arrays.asList(line.split(" "));
  }
});
words.first(); // 返回"hello"
```

我们在图 3-3 中阐释了 `flatMap()` 和 `map()` 的区别。你可以把 `flatMap()` 看作将返回的迭代器"拍扁"，这样就得到了一个由各列表中的元素组成的 RDD，而不是一个由列表组成的 RDD。

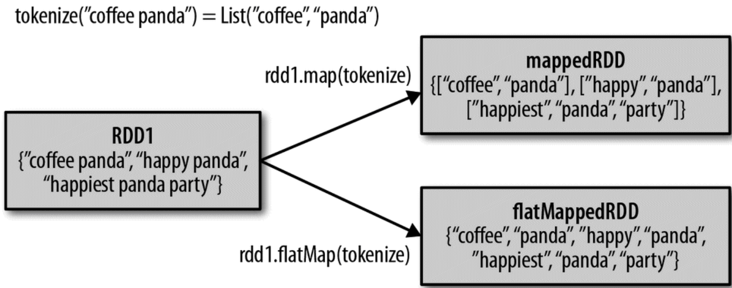


图 **3-3**：**RDD** 的 **`flatMap()`** 和 **`map()`** 的区别

2. 伪集合操作

尽管 RDD 本身不是严格意义上的集合，但它也支持许多数学上的集合操作，比如合并和相交操作。图 3-4 展示了四种操作。注意，这些操作都要求操作的 RDD 是相同数据类型的。

我们的 RDD 中最常缺失的集合属性是元素的唯一性，因为常常有重复的元素。如果只要唯一的元素，我们可以使用 `RDD.distinct()` 转化操作来生成一个只包含不同元素的新 RDD。不过需要注意，`distinct()` 操作的开销很大，因为它需要将所有数据通过网络进行混洗（shuffle），以确保每个元素都只有一份。第 4 章会详细介绍数据混洗，以及如何避免数据混洗。

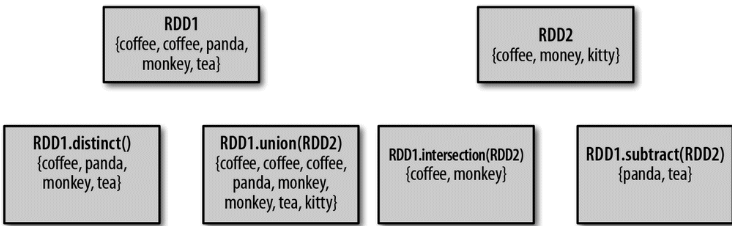


图 **3-4**：一些简单的集合操作

最简单的集合操作是 `union(other)`，它会返回一个包含两个 RDD

中所有元素的 RDD。这在很多用例下都很有用，比如处理来自多个数据源的日志文件。与数学中的 `union()` 操作不同的是，如果输入的 RDD 中有重复数据，Spark 的 `union()` 操作也会包含这些重复数据（如有必要，我们可以通过 `distinct()` 实现相同的效果）。

Spark 还提供了 `intersection(other)` 方法，只返回两个 RDD 中都有的元素。`intersection()` 在运行时也会去掉所有重复的元素（单个 RDD 内的重复元素也会一起移除）。尽管 `intersection()` 与 `union()` 的概念相似，`intersection()` 的性能却要差很多，因为它需要通过网络混洗数据来发现共有的元素。

有时我们需要移除一些数据。`subtract(other)` 函数接收另一个 RDD作为参数，返回一个由只存在于第一个 RDD 中而不存在于第二个 RDD 中的所有元素组成的 RDD。和 `intersection()` 一样，它也需要数据混洗。

我们也可以计算两个 RDD 的笛卡儿积，如图 3-5 所示。`cartesian(other)` 转化操作会返回所有可能的 `(a, b)` 对，其中 `a` 是源 RDD 中的元素，而 `b` 则来自另一个 RDD。笛卡儿积在我们希望考虑所有可能的组合的相似度时比较有用，比如计算各用户对各种产品的预期兴趣程度。我们也可以求一个 RDD 与其自身的笛卡儿积，这可以用于求用户相似度的应用中。不过要特别注意的是，求大规模 RDD 的笛卡儿积开销巨大。

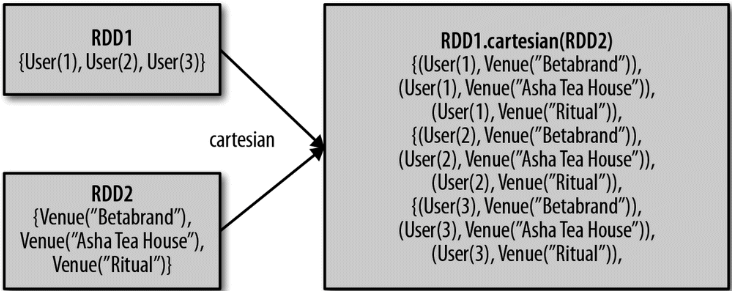


图 **3-5**：两个 **RDD** 的笛卡儿积

表 3-2 和表 3-3 总结了这些常见的 RDD 转化操作。

表3-2：对一个数据为{1, 2, 3, 3}的RDD进行基本的RDD转化操作

函数名	目的
map()	将函数应用于 RDD 中的每个元素，将返回值构成新的 RDD
flatMap()	将函数应用于 RDD 中的每个元素，将返回的迭代器的所有内容构成新的 RDD。通常用来切分单词
filter()	返回一个由通过传给 filter() 的函数的元素组成的 RDD
distinct()	去重
sample(withReplacement, fraction, [seed])	对 RDD 采样，以及是否替换

表3-3：对数据分别为{1, 2, 3}和{3, 4, 5}的RDD进行针对两个RDD的转化操作

函数名	目的
union()	生成一个包含两个 RDD 中所有元素的 RDD
intersection()	求两个 RDD 共同的元素的 RDD
subtract()	移除一个 RDD 中的内容（例如移除训练数据）
cartesian()	与另一个 RDD 的笛卡儿积

3. 行动操作

你很有可能会用到基本 RDD 上最常见的行动操作 `reduce()`。它接收一个函数作为参数，这个函数要操作两个相同元素类型的 RDD 数据并返回一个同样类型的新元素。一个简单的例子就是函数 +，可以用它来对我们的 RDD 进行累加。使用 `reduce()`，可以很方便地计算出 RDD 中所有元素的总和、元素的个数，以及其他类型的聚合操作（如例 3-32 至例 3-34 所示）。

例 **3-32**：Python 中的 `reduce()`

```
sum = rdd.reduce(lambda x, y: x + y)
```

例 **3-33**：Scala 中的 `reduce()`

```
val sum = rdd.reduce((x, y) => x + y)
```

例 **3-34**：Java 中的 `reduce()`

```
Integer sum = rdd.reduce(new Function2<Integer, Integer, Integer>() {
  public Integer call(Integer x, Integer y) { return x + y; }
});
```

fold() 和 reduce() 类似，接收一个与 reduce() 接收的函数签名相同的函数，再加上一个“初始值”来作为每个分区第一次调用时的结果。你所提供的初始值应当是你提供的操作的单位元素；也就是说，使用你的函数对这个初始值进行多次计算不会改变结果（例如 + 对应的 0，* 对应的 1，或拼接操作对应的空列表）。


我们可以通过原地修改并返回两个参数中的前一个的值来节约在 fold() 中创建对象的开销。但是你没有办法修改第二个参数。

fold() 和 reduce() 都要求函数的返回值类型需要和我们所操作的 RDD 中的元素类型相同。这很符合像 sum 这种操作的情况。但有时我们确实需要返回一个不同类型的值。例如，在计算平均值时，需要记录遍历过程中的计数以及元素的数量，这就需要我们返回一个二元组。可以先对数据使用 map() 操作，来把元素转为该元素和 1 的二元组，也就是我们所希望的返回类型。这样 reduce() 就可以以二元组的形式进行归约了。

aggregate() 函数则把我们从返回值类型必须与所操作的 RDD 类型相同的限制中解放出来。与 fold() 类似，使用 aggregate() 时，需要提供我们期待返回的类型的初始值。然后通过一个函数把 RDD 中的元素合并起来放入累加器。考虑到每个节点是在本地进行累加的，最终，还需要提供第二个函数来将累加器两两合并。

我们可以用 aggregate() 来计算 RDD 的平均值，来代替 map() 后面接 fold() 的方式，如例 3-35 至例 3-37 所示。

例 **3-35**：Python 中的 aggregate()

```
sumCount = nums.aggregate((0, 0),
               (lambda acc, value: (acc[0] + value, acc[1] + 1),
               (lambda acc1, acc2: (acc1[0] + acc2[0], acc1[1] + acc2[1]))))
return sumCount[0] / float(sumCount[1])
```

例 **3-36**：Scala 中的 aggregate()

```
val result = input.aggregate((0, 0))(
               (acc, value) => (acc._1 + value, acc._2 + 1),
               (acc1, acc2) => (acc1._1 + acc2._1, acc1._2 + acc2._2))
val avg = result._1 / result._2.toDouble
```

例 3-37：Java 中的 aggregate()

```
class AvgCount implements Serializable {
  public AvgCount(int total, int num) {
    this.total = total;
    this.num = num;
  }
  public int total;
  public int num;
  public double avg() {
    return total / (double) num;
  }
}
Function2<AvgCount, Integer, AvgCount> addAndCount =
  new Function2<AvgCount, Integer, AvgCount>() {
    public AvgCount call(AvgCount a, Integer x) {
      a.total += x;
      a.num += 1;
      return a;
  }
};
Function2<AvgCount, AvgCount, AvgCount> combine =
  new Function2<AvgCount, AvgCount, AvgCount>() {
  public AvgCount call(AvgCount a, AvgCount b) {
    a.total += b.total;
    a.num += b.num;
    return a;
  }
};
AvgCount initial = new AvgCount(0, 0);
AvgCount result = rdd.aggregate(initial, addAndCount, combine);
System.out.println(result.avg());
```

RDD 的一些行动操作会以普通集合或者值的形式将 RDD 的部分或全部数据返回驱动器程序中。

把数据返回驱动器程序中最简单、最常见的操作是 collect()，它会将整个 RDD 的内容返回。collect() 通常在单元测试中使用，因为此时 RDD 的整个内容不会很大，可以放在内存中。使用 collect() 使得 RDD 的值与预期结果之间的对比变得很容易。由于需要将数据复制到驱动器进程中，collect() 要求所有数据都必须能一同放入单台机器的内存中。

take(n) 返回 RDD 中的 n 个元素，并且尝试只访问尽量少的分区，因此该操作会得到一个不均衡的集合。需要注意的是，这些操作返回元素的顺序与你预期的可能不一样。

这些操作对于单元测试和快速调试都很有用，但是在处理大规模数据时会遇到瓶颈。

如果为数据定义了顺序，就可以使用 `top()` 从 RDD 中获取前几个元素。`top()` 会使用数据的默认顺序，但我们也可以提供自己的比较函数，来提取前几个元素。

有时需要在驱动器程序中对我们的数据进行采样。`takeSample(withReplacement, num, seed)` 函数可以让我们从数据中获取一个采样，并指定是否替换。

有时我们会对 RDD 中的所有元素应用一个行动操作，但是不把任何结果返回到驱动器程序中，这也是有用的。比如可以用 JSON 格式把数据发送到一个网络服务器上，或者把数据存到数据库中。不论哪种情况，都可以使用 `foreach()` 行动操作来对 RDD 中的每个元素进行操作，而不需要把 RDD 发回本地。

关于基本 RDD 上的更多标准操作，我们都可以从其名称推测出它们的行为。`count()` 用来返回元素的个数，而 `countByValue()` 则返回一个从各值到值对应的计数的映射表。表 3-4 总结了这些行动操作。

表3-4：对一个数据为{1, 2, 3, 3}的RDD进行基本的RDD行动操作

函数名	目的	示例	结果
`collect()`	返回 RDD 中的所有元素	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	RDD 中的元素个数	`rdd.count()`	4
`countByValue()`	各元素在 RDD 中出现的次数	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
`take(num)`	从 RDD 中返回 num 个元素	`rdd.take(2)`	`{1, 2}`
`top(num)`	从 RDD 中返回最前面的 num 个元素	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	从 RDD 中按照提供的顺序返回最前面的 num 个元素	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`takeSample(withReplacement, num, [seed])`	从 RDD 中返回任意一些元素	`rdd.takeSample(false, 1)`	非确定的
`reduce(func)`	并行整合 RDD 中所有数据（例如 sum）	`rdd.reduce((x, y) => x + y)`	9
`fold(zero)(func)`	和 reduce() 一样，但是需要提供初始值	`rdd.fold(0)((x, y) => x + y)`	9
`aggregate(zeroValue)(seqOp, combOp)`	和 reduce() 相似，但是通常返回不同类型的函数	`rdd.aggregate((0, 0))((x, y) => (x._1 + y, x._2 + 1), (x, y) => (x._1 + y._1, x._2 + y._2))`	`(9, 4)`
`foreach(func)`	对 RDD 中的每个元素使用给定的函数	`rdd.foreach(func)`	无

3.5.2　在不同RDD类型间转换

有些函数只能用于特定类型的 RDD，比如 `mean()` 和

`variance()` 只能用在数值 RDD 上，而 `join()` 只能用在键值对 RDD 上。我们会在第 6 章讨论数值 RDD 的专门函数，在第 4 章讨论键值对 RDD 的专有操作。在 Scala 和 Java 中，这些函数都没有定义在标准的 RDD 类中，所以要访问这些附加功能，必须要确保获得了正确的专用 RDD 类。

1. Scala

在 Scala 中，将 RDD 转为有特定函数的 RDD（比如在 `RDD[Double]` 上进行数值操作）是由隐式转换来自动处理的。2.4.1 节中提到过，我们需要加上 `import org.apache.spark.SparkContext._` 来使用这些隐式转换。你可以在 SparkContext 对象的 Scala 文档（http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.SparkContext$）中查看所列出的隐式转换。这些隐式转换可以隐式地将一个 RDD 转为各种封装类，比如 `DoubleRDDFunctions`（数值数据的 RDD）和 `PairRDDFunctions`（键值对 RDD），这样我们就有了诸如 `mean()` 和 `variance()` 之类的额外的函数。

隐式转换虽然强大，但是会让阅读代码的人感到困惑。如果你对 RDD 调用了像 `mean()` 这样的函数，可能会发现 RDD 类的 Scala 文档（http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.rdd.RDD）中根本没有 `mean()` 函数。调用之所以能够成功，是因为隐式转换可以把 `RDD[Double]` 转为 `DoubleRDDFunctions`。当我们在 Scala 文档中查找函数时，不要忘了那些封装的专用类中的函数。

2. Java

在 Java 中，各种 RDD 的特殊类型间的转换更为明确。Java 中有两个专门的类 `JavaDoubleRDD` 和 `JavaPairRDD`，来处理特殊类型的 RDD，这两个类还针对这些类型提供了额外的函数。这让你可以更加了解所发生的一切，但是也显得有些累赘。

要构建出这些特殊类型的 RDD，需要使用特殊版本的类来替代一般使用的 `Function` 类。如果要从 `T` 类型的 RDD 创建出一个 `JavaDoubleRDD`，我们就应当在映射操作中使用 `DoubleFunction<T>` 来替代 `Function<T, Double>`。表 3-5 展示了一些特殊版本的函数类及其用法。

此外，我们也需要调用 RDD 上的一些别的函数（因此不能只是创建

出一个 DoubleFunction 然后把它传给 map()）。当需要一个 JavaDoubleRDD 时，我们应当调用 mapToDouble() 来替代 map()，跟其他所有函数所遵循的模式一样。

表3-5：**Java**中针对专门类型的函数接口

函数名	等价函数	用途
DoubleFlatMapFunction<T>	Function<T, Iterable<Double>>	用于 flatMapToDouble，以生成 DoubleRDD
DoubleFunction<T>	Function<T, Double>	用于 mapToDouble，以生成 DoubleRDD
PairFlatMapFunction<T, K, V>	Function<T, Iterable<Tuple2<K, V>>>	用于 flatMapToPair，以生成 PairRDD<K, V>
PairFunction<T, K, V>	Function<T, Tuple2<K, V>>	用于 mapToPair，以生成 PairRDD<K, V>

我们可以把例 3-28 修改为生成一个 JavaDoubleRDD、计算 RDD 中每个元素的平方值的示例，如例 3-38 所示。这样就可以调用 DoubleRDD 独有的函数了，比如 mean() 和 variance()。

例 **3-38**：用 Java 创建 DoubleRDD

```
JavaDoubleRDD result = rdd.mapToDouble(
  new DoubleFunction<Integer>() {
    public double call(Integer x) {
      return (double) x * x;
    }
});
System.out.println(result.mean());
```

3. Python

Python 的 API 结构与 Java 和 Scala 有所不同。在 Python 中，所有的函数都实现在基本的 RDD 类中，但如果操作对应的 RDD 数据类型不正确，就会导致运行时错误。

3.6　持久化(缓存)

如前所述，Spark RDD 是惰性求值的，而有时我们希望能多次使用同一个 RDD。如果简单地对 RDD 调用行动操作，Spark 每次都会重算 RDD 以及它的所有依赖。这在迭代算法中消耗格外大，因为迭代算法常常会多次使用同一组数据。例 3-39 就是先对 RDD 作一次计数、再把该 RDD 输出的一个小例子。

例 **3-39**：Scala 中的两次执行

```
val result = input.map(x => x*x)
println(result.count())
println(result.collect().mkString(","))
```

为了避免多次计算同一个 RDD，可以让 Spark 对数据进行持久化。当我们让 Spark 持久化存储一个 RDD 时，计算出 RDD 的节点会分别保存它们所求出的分区数据。如果一个有持久化数据的节点发生故障，Spark 会在需要用到缓存的数据时重算丢失的数据分区。如果希望节点故障的情况不会拖累我们的执行速度，也可以把数据备份到多个节点上。

出于不同的目的，我们可以为 RDD 选择不同的持久化级别（如表 3-6 所示）。在 Scala（见例 3-40）和 Java 中，默认情况下 `persist()` 会把数据以序列化的形式缓存在 JVM 的堆空间中。在 Python 中，我们会始终序列化要持久化存储的数据，所以持久化级别默认值就是以序列化后的对象存储在 JVM 堆空间中。当我们把数据写到磁盘或者堆外存储上时，也总是使用序列化后的数据。

表3-6：**`org.apache.spark.storage.StorageLevel`**和**`pyspark.StorageLevel`**中的持久化级别；如有必要，可以通过在存储级别的末尾加上“**_2**”来把持久化数据存为两份

[illegible]									
EMORY_ONLY									
EMORY_ONLY_SER									
[illegible] 如果数据在内存中放不下，则溢写到磁盘上									
[illegible] 如果数据在内存中放不下，则溢写到磁盘上。在内存中存放序列化后的数据									
SK_ONLY									



堆外缓存是试验性功能，我们使用 Tachyon（http://tachyon-project.org/）作为外部系统。如果你对 Spark 的堆外缓存有兴趣，可以看看关于如何在 Tachyon 上运行 Spark 的介绍（http://tachyon-project.org/Running-Spark-on-Tachyon.html）。

例 **3-40**：在 Scala 中使用 `persist()`

```
import org.apache.spark.storage.StorageLevel
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
```

```
println(result.count())
println(result.collect().mkString(","))
```

注意，我们在第一次对这个 RDD 调用行动操作前就调用了 persist() 方法。persist() 调用本身不会触发强制求值。

如果要缓存的数据太多，内存中放不下，Spark 会自动利用最近最少使用（LRU）的缓存策略把最老的分区从内存中移除。对于仅把数据存放在内存中的缓存级别，下一次要用到已经被移除的分区时，这些分区就需要重新计算。但是对于使用内存与磁盘的缓存级别的分区来说，被移除的分区都会写入磁盘。不论哪一种情况，都不必担心你的作业因为缓存了太多数据而被打断。不过，缓存不必要的数据会导致有用的数据被移出内存，带来更多重算的时间开销。

最后，RDD 还有一个方法叫作 unpersist()，调用该方法可以手动把持久化的 RDD 从缓存中移除。

3.7 总结

在本章中，我们介绍了 RDD 运行模型以及 RDD 的许多常见操作。如果你读到了这里，恭喜——你已经学完了 Spark 的所有核心概念。我们在进行并行聚合、分组等操作时，常常需要利用键值对形式的 RDD。下一章会讲解键值对形式的 RDD 上一些相关的特殊操作。然后，我们会讨论各种数据源的输入输出，以及一些关于使用 SparkContext 的进阶话题。

第 4 章　键值对操作

键值对 RDD 是 Spark 中许多操作所需要的常见数据类型。本章就来介绍如何操作键值对 RDD。键值对 RDD 通常用来进行聚合计算。我们一般要先通过一些初始 ETL（抽取、转化、装载）操作来将数据转化为键值对形式。键值对 RDD 提供了一些新的操作接口（比如统计每个产品的评论，将数据中键相同的分为一组，将两个不同的 RDD 进行分组合并等）。

本章也会讨论用来让用户控制键值对 RDD 在各节点上分布情况的高级特性：分区。有时，使用可控的分区方式把常被一起访问的数据放到同一个节点上，可以大大减少应用的通信开销。这会带来明显的性能提升。我们会使用 PageRank 算法来演示分区的作用。为分布式数据集选择正确的分区方式和为本地数据集选择合适的数据结构很相似——在这两种情况下，数据的分布都会极其明显地影响程序的性能表现。

4.1　动机

Spark 为包含键值对类型的 RDD 提供了一些专有的操作。这些 RDD 被称为 pair RDD1。Pair RDD 是很多程序的构成要素，因为它们提供了并行操作各个键或跨节点重新进行数据分组的操作接口。例如，pair RDD 提供 `reduceByKey()` 方法，可以分别归约每个键对应的数据，还有 `join()` 方法，可以把两个 RDD 中键相同的元素组合到一起，合并为一个 RDD。我们通常从一个 RDD 中提取某些字段（例如代表事件时间、用户 ID 或者其他标识符的字段），并使用这些字段作为 pair RDD 操作中的键。

> 1“pair RDD”意为“对 RDD”，为避免引发歧义，译文中保留“pair RDD”。——译者注

4.2　创建Pair RDD

在 Spark 中有很多种创建 pair RDD 的方式。第 5 章会讲到，很多存储键值对的数据格式会在读取时直接返回由其键值对数据组成的 pair RDD。此外，当需要把一个普通的 RDD 转为 pair RDD 时，可以调用 `map()` 函数来实现，传递的函数需要返回键值对。后面会展示如何将

由文本行组成的 RDD 转换为以每行的第一个单词为键的 pair RDD。

构建键值对 RDD 的方法在不同的语言中会有所不同。在 Python 中，为了让提取键之后的数据能够在函数中使用，需要返回一个由二元组组成的 RDD（见例 4-1）。

例 **4-1**：在 Python 中使用第一个单词作为键创建出一个 pair RDD

```
pairs = lines.map(lambda x: (x.split(" ")[0], x))
```

在 Scala 中，为了让提取键之后的数据能够在函数中使用，同样需要返回二元组（见例 4-2）。隐式转换可以让二元组 RDD 支持附加的键值对函数。

例 **4-2**：在 Scala 中使用第一个单词作为键创建出一个 pair RDD

```
val pairs = lines.map(x => (x.split(" ")(0), x))
```

Java 没有自带的二元组类型，因此 Spark 的 Java API 让用户使用 `scala.Tuple2` 类来创建二元组。这个类很简单：Java 用户可以通过 `new Tuple2(elem1, elem2)` 来创建一个新的二元组，并且可以通过 `._1()` 和 `._2()` 方法访问其中的元素。

Java 用户还需要调用专门的 Spark 函数来创建 pair RDD。例如，要使用 `mapToPair()` 函数来代替基础版的 `map()` 函数，这在 3.5.2 节中的“Java”一节有过更详细的讨论。下面通过例 4-3 中展示一个简单的例子。

例 **4-3**：在 Java 中使用第一个单词作为键创建出一个 pair RDD

```
PairFunction<String, String, String> keyData =
  new PairFunction<String, String, String>() {
  public Tuple2<String, String> call(String x) {
    return new Tuple2(x.split(" ")[0], x);
  }
};
JavaPairRDD<String, String> pairs = lines.mapToPair(keyData);
```

当用 Scala 和 Python 从一个内存中的数据集创建 pair RDD 时，只需要对这个由二元组组成的集合调用 `SparkContext.parallelize()` 方法。而要使用 Java 从内存数据集创建 pair RDD 的话，则需要使用 `SparkContext.parallelizePairs()`。

4.3　Pair RDD的转化操作

Pair RDD 可以使用所有标准 RDD 上的可用的转化操作。3.4 节中介绍的所有有关传递函数的规则也都同样适用于 pair RDD。由于 pair RDD 中包含二元组，所以需要传递的函数应当操作二元组而不是独立的元素。表 4-1 和表 4-2 总结了对 pair RDD 的一些转化操作，在本章会深入介绍。

表4-1：Pair RDD的转化操作（以键值对集合`{(1, 2), (3, 4), (3, 6)}`为例）

[illegible]			
合并具有相同键的值	[illegible] v) => x + y)		
对具有相同键的值进行分组	[illegible]		
使用不同的返回类型合并具有相同键的值	[illegible] value, mergeCombiners, partitioner)		
对pair RDD中的每个值应用一个函数而不改变键	[illegible]		
对pair RDD中的每个值应用一个返回迭代器的函数，然后对返回的每个元素都生成一个对应原键的键值对记录。通常用于符号化	[illegible] (3,5))		
返回一个仅包含键的RDD	[illegible]		
返回一个仅包含值的RDD	[illegible]		
返回一个根据键排序的RDD	[illegible]		

2在Scala 中此处不应使用括号。——译者注

3在Scala 中此处不应使用括号。——译者注

表4-2：针对两个pair RDD的转化操作（rdd = `{(1, 2), (3, 4), (3, 6)}`other = `{(3, 9)}`）

[illegible]			
删掉 RDD 中键与 other RDD 中的键相同的元素	[illegible]		
对两个 RDD 进行内连接	[illegible] 9))}		
对两个 RDD 进行连接操作，确保第一个 RDD 的键必须存在（右外连接）	[illegible]		
对两个 RDD 进行连接操作，确保第二个 RDD 的键必须存在（左外连接）	[illegible]		
将两个 RDD 中拥有相同键的数据分组到一起	[illegible]		

接下来的几节会详细探讨这些 pair RDD 的函数。

Pair RDD 也还是 RDD（元素为 Java 或 Scala 中的 Tuple2 对象或 Python 中的元组），因此同样支持 RDD 所支持的函数。例如，我们可以拿前一节中的 pair RDD，筛选掉长度超过 20 个字符的行，如例 4-4 至例 4-6 以及图 4-1 所示。

例 **4-4**：用 Python 对第二个元素进行筛选

```
result = pairs.filter(lambda keyValue: len(keyValue[1]) < 20)
```

例 **4-5**：用 Scala 对第二个元素进行筛选

```
pairs.filter{case (key, value) => value.length < 20}
```

例 **4-6**：用 Java 对第二个元素进行筛选

```
Function<Tuple2<String, String>, Boolean> longWordFilter =
  new Function<Tuple2<String, String>, Boolean>() {
    public Boolean call(Tuple2<String, String> keyValue) {
      return (keyValue._2().length() < 20);
    }
  };
JavaPairRDD<String, String> result = pairs.filter(longWordFilter);
```

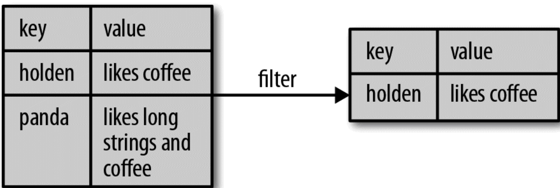


图 **4-1**：根据值筛选

有时，我们只想访问 pair RDD 的值部分，这时操作二元组很麻烦。由于这是一种常见的使用模式，因此 Spark 提供了 `mapValues(func)` 函数，功能类似于 `map{case (x, y): (x, func(y))}`。可以在很多例子中使用这个函数。

接下来就依次讨论 pair RDD 的各种操作，先从聚合操作开始。

4.3.1 聚合操作

当数据集以键值对形式组织的时候，聚合具有相同键的元素进行一些统计是很常见的操作。之前讲解过基础 RDD 上的 `fold()`、`combine()`、`reduce()` 等行动操作，pair RDD 上则有相应的针对键的转化操作。Spark 有一组类似的操作，可以组合具有相同键的值。这些操作返回 RDD，因此它们是转化操作而不是行动操作。

`reduceByKey()` 与 `reduce()` 相当类似；它们都接收一个函数，并使用该函数对值进行合并。`reduceByKey()` 会为数据集中的每个键进行并行的归约操作，每个归约操作会将键相同的值合并起来。因为数据集中可能有大量的键，所以 `reduceByKey()` 没有被实现为向用户程序返回一个值的行动操作。实际上，它会返回一个由各键和对应键归约出来的结果值组成的新的 RDD。

`foldByKey()` 则与 `fold()` 相当类似；它们都使用一个与 RDD 和合并函数中的数据类型相同的零值作为初始值。与 `fold()` 一样，`foldByKey()` 操作所使用的合并函数对零值与另一个元素进行合并，结果仍为该元素。

如例 4-7 和例 4-8 所示，可以使用 `reduceByKey()` 和 `mapValues()` 来计算每个键的对应值的均值（见图 4-2）。这和使用 `fold()` 和 `map()` 计算整个 RDD 平均值的过程很相似。对于求平均，可以使用更加专用的函数来获取同样的结果，后面就会讲到。

例 **4-7**：在 Python 中使用 `reduceByKey()` 和 `mapValues()` 计算每个键对应的平均值

```
rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda x, y: (x[0] + y[0], x[1
```

例 **4-8**：在 Scala 中使用 `reduceByKey()` 和 `mapValues()` 计算每个键对应的平均值

```
rdd.mapValues(x => (x, 1)).reduceByKey((x, y) => (x._1 + y._1, x._2 + y._2
```

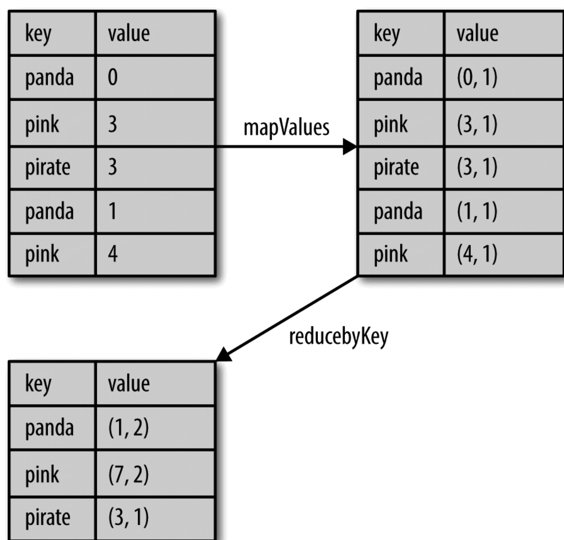


图 **4-2**：求每个键平均值的数据流

熟悉 MapReduce 中的合并器（combiner）概念的读者可能已经注意到，调用 `reduceByKey()` 和 `foldByKey()` 会在为每个键计算全局的总结果之前先自动在每台机器上进行本地合并。用户不需要指定合并器。更泛化的 `combineByKey()` 接口可以让你自定义合并的行为。

我们也可以使用例 4-9 到例 4-11 中展示的方法来解决经典的分布式单词计数问题。可以使用前一章中讲过的 `flatMap()` 来生成以单词为键、以数字 1 为值的 pair RDD，然后像例 4-7 和例 4-8 中那样，使用 `reduceByKey()` 对所有的单词进行计数。

例 **4-9**：用 Python 实现单词计数

```
rdd = sc.textFile("s3://...")
words = rdd.flatMap(lambda x: x.split(" "))
result = words.map(lambda x: (x, 1)).reduceByKey(lambda x, y: x + y)
```

例 **4-10**：用 Scala 实现单词计数

```
val input = sc.textFile("s3://...")
val words = input.flatMap(x => x.split(" "))
```

```
val result = words.map(x => (x, 1)).reduceByKey((x, y) => x + y)
```

例 **4-11**：用 Java 实现单词计数

```
JavaRDD<String> input = sc.textFile("s3://...")
JavaRDD<String> words = rdd.flatMap(new FlatMapFunction<String, String>()
  public Iterable<String> call(String x) { return Arrays.asList(x.split("
});
JavaPairRDD<String, Integer> result = words.mapToPair(
  new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String x) { return new Tuple2(x, 1
}).reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer a, Integer b) { return a + b; }
});
```



事实上，我们可以对第一个 RDD 使用 `countByValue()` 函数，以更快地实现单词计数：`input.flatMap(x => x.split(" ")).countByValue()`。

`combineByKey()` 是最为常用的基于键进行聚合的函数。大多数基于键聚合的函数都是用它实现的。和 `aggregate()` 一样，`combineByKey()` 可以让用户返回与输入数据的类型不同的返回值。

要理解 `combineByKey()`，要先理解它在处理数据时是如何处理每个元素的。由于 `combineByKey()` 会遍历分区中的所有元素，因此每个元素的键要么还没有遇到过，要么就和之前的某个元素的键相同。

如果这是一个新的元素，`combineByKey()` 会使用一个叫作 `createCombiner()` 的函数来创建那个键对应的累加器的初始值。需要注意的是，这一过程会在每个分区中第一次出现各个键时发生，而不是在整个 RDD 中第一次出现一个键时发生。

如果这是一个在处理当前分区之前已经遇到的键，它会使用 `mergeValue()` 方法将该键的累加器对应的当前值与这个新的值进行合并。

由于每个分区都是独立处理的，因此对于同一个键可以有多个累加器。如果有两个或者更多的分区都有对应同一个键的累加器，就需要

使用用户提供的 mergeCombiners() 方法将各个分区的结果进行合并。

 如果已知数据在进行 combineByKey() 时无法从 map 端聚合中获益的话，可以禁用它。例如，由于聚合函数（追加到一个队列）无法在 map 端聚合时节约任何空间，groupByKey() 就把它禁用了。如果希望禁用 map 端组合，就需要指定分区方式。就目前而言，你可以通过传递 rdd.partitioner 来直接使用源 RDD 的分区方式。

combineByKey() 有多个参数分别对应聚合操作的各个阶段，因而非常适合用来解释聚合操作各个阶段的功能划分。为了更好地演示 combineByKey() 是如何工作的，下面来看看如何计算各键对应的平均值，如例 4-12 至例 4-14 和图 4-3 所示。

例 **4-12**：在 Python 中使用 combineByKey() 求每个键对应的平均值

```
sumCount = nums.combineByKey((lambda x: (x,1)),
                             (lambda x, y: (x[0] + y, x[1] + 1)),
                             (lambda x, y: (x[0] + y[0], x[1] + y[1])))
sumCount.map(lambda key, xy: (key, xy[0]/xy[1])).collectAsMap()
```

例 **4-13**：在 Scala 中使用 combineByKey() 求每个键对应的平均值

```
val result = input.combineByKey(
  (v) => (v, 1),
  (acc: (Int, Int), v) => (acc._1 + v, acc._2 + 1),
  (acc1: (Int, Int), acc2: (Int, Int)) => (acc1._1 + acc2._1, acc1._2 + ac
  ).map{ case (key, value) => (key, value._1 / value._2.toFloat) }
  result.collectAsMap().map(println(_))
```

例 **4-14**：在 Java 中使用 combineByKey() 求每个键对应的平均值

```
public static class AvgCount implements Serializable {
  public AvgCount(int total, int num) {   total_ = total;  num_ = num; }
  public int total_;
  public int num_;
```

```
  public float avg() {   returntotal_/(float)num_; }
}

Function<Integer, AvgCount> createAcc = new Function<Integer, AvgCount>() {
  public AvgCount call(Integer x) {
    return new AvgCount(x, 1);
  }
};
Function2<AvgCount, Integer, AvgCount> addAndCount =
  new Function2<AvgCount, Integer, AvgCount>() {
  public AvgCount call(AvgCount a, Integer x) {
    a.total_ += x;
    a.num_ += 1;
    return a;
  }
};
Function2<AvgCount, AvgCount, AvgCount> combine =
  new Function2<AvgCount, AvgCount, AvgCount>() {
  public AvgCount call(AvgCount a, AvgCount b) {
    a.total_ += b.total_;
    a.num_ += b.num_;
    return a;
  }
};
AvgCount initial = new AvgCount(0,0);
JavaPairRDD<String, AvgCount> avgCounts =
  nums.combineByKey(createAcc, addAndCount, combine);
Map<String, AvgCount> countMap = avgCounts.collectAsMap();
for (Entry<String, AvgCount> entry : countMap.entrySet()) {
  System.out.println(entry.getKey() + ":" + entry.getValue().avg());
}
```

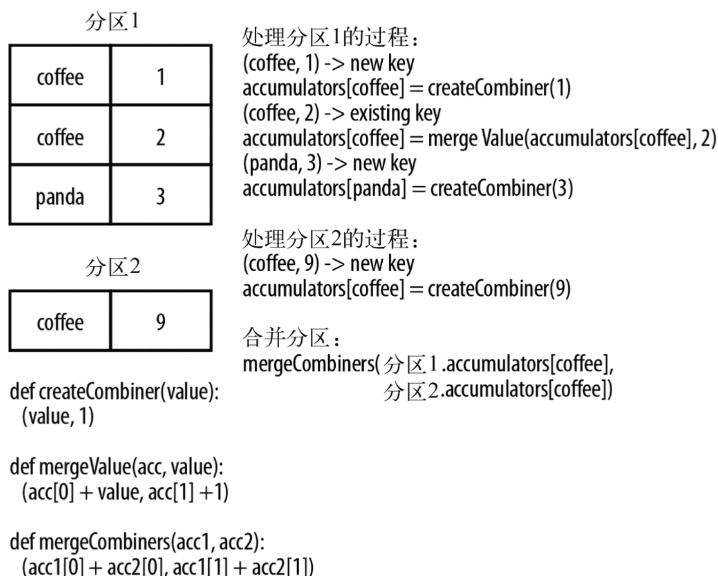


图 **4-3**：**`combineByKey()`** 数据流示意图

有很多函数可以进行基于键的数据合并。它们中的大多数都是在 `combineByKey()` 的基础上实现的，为用户提供了更简单的接口。不管怎样，在 Spark 中使用这些专用的聚合函数，始终要比手动将数据分组再归约快很多。

并行度调优

到目前为止，我们已经讨论了所有的转化操作的分发方式，但是还没有探讨 Spark 是怎样确定如何分割工作的。每个 RDD 都有固定数目的分区，分区数决定了在 RDD 上执行操作时的并行度。

在执行聚合或分组操作时，可以要求 Spark 使用给定的分区数。Spark 始终尝试根据集群的大小推断出一个有意义的默认值，但是有时候你可能要对并行度进行调优来获取更好的性能表现。

本章讨论的大多数操作符都能接收第二个参数，这个参数用来指定分组结果或聚合结果的RDD 的分区数，如例 4-15 和例 4-16 所示。

例 **4-15**：在 Python 中自定义 `reduceByKey()` 的并行度

```
data = [("a", 3), ("b", 4), ("a", 1)]
```

```
sc.parallelize(data).reduceByKey(lambda x, y: x + y)      # 默认并行度
sc.parallelize(data).reduceByKey(lambda x, y: x + y, 10)  # 自定义并行度
```

例 4-16：在 Scala 中自定义 `reduceByKey()` 的并行度

```
val data = Seq(("a", 3), ("b", 4), ("a", 1))
sc.parallelize(data).reduceByKey((x, y) => x + y)    // 默认并行度
sc.parallelize(data).reduceByKey((x, y) => x + y, 10)    // 自定义并行度
```

有时，我们希望在除分组操作和聚合操作之外的操作中也能改变 RDD 的分区。对于这样的情况，Spark 提供了 `repartition()` 函数。它会把数据通过网络进行混洗，并创建出新的分区集合。切记，对数据进行重新分区是代价相对比较大的操作。Spark 中也有一个优化版的 `repartition()`，叫作 `coalesce()`。你可以使用 Java 或 Scala 中的 `rdd.partitions.size` 以及 Python 中的 `rdd.getNumPartitions` 查看 RDD 的分区数，并确保调用 `coalesce()` 时将 RDD 合并到比现在的分区数更少的分区中。

4.3.2　数据分组

对于有键的数据，一个常见的用例是将数据根据键进行分组——比如查看一个顾客的所有订单。

如果数据已经以预期的方式提取了键，`groupByKey()` 就会使用 RDD 中的键来对数据进行分组。对于一个由类型 `K` 的键和类型 `V` 的值组成的 RDD，所得到的结果 RDD 类型会是 `[K, Iterable[V]]`。

`groupByKey()` 可以用于未成对的数据上，也可以根据除键相同以外的条件进行分组。它可以接收一个函数，对源 RDD 中的每个元素使用该函数，将返回结果作为键再进行分组。


如果你发现自己写出了先使用 `groupByKey()` 然后再对值使用 `reduce()` 或者 `fold()` 的代码，你很有可能可以通过使用一种根据键进行聚合的函数来更高效地实现同样的效果。对每个键归约数据，返回对应每个键的归约值的 RDD，而不是把 RDD 归约为一个内存中的值。例如，`rdd.reduceByKey(func)` 与

> rdd.groupByKey().mapValues(value => value.reduce(func)) 等价，但是前者更为高效，因为它避免了为每个键创建存放值的列表的步骤。

除了对单个 RDD 的数据进行分组，还可以使用一个叫作 cogroup() 的函数对多个共享同一个键的 RDD 进行分组。对两个键的类型均为 K 而值的类型分别为 V 和 W 的 RDD 进行 cogroup() 时，得到的结果 RDD 类型为 [(K, (Iterable[V], Iterable[W]))]。如果其中的一个 RDD 对于另一个 RDD 中存在的某个键没有对应的记录，那么对应的迭代器则为空。cogroup() 提供了为多个 RDD 进行数据分组的方法。

cogroup() 是下一节中要讲的连接操作的构成要素。

> 
> cogroup() 不仅可以用于实现连接操作，还可以用来求键的交集。除此之外，cogroup() 还能同时应用于三个及以上的 RDD。

4.3.3　连接

将有键的数据与另一组有键的数据一起使用是对键值对数据执行的最有用的操作之一。连接数据可能是 pair RDD 最常用的操作之一。连接方式多种多样：右外连接、左外连接、交叉连接以及内连接。

普通的 join 操作符表示内连接 [4]。只有在两个 pair RDD 中都存在的键才叫输出。当一个输入对应的某个键有多个值时，生成的 pair RDD 会包括来自两个输入 RDD 的每一组相对应的记录。例 4-17 可以帮你理解这个定义。

> 4“连接”是数据库术语，表示将两张表根据相同的值来组合字段。

例 4-17：在 Scala shell 中进行内连接

```
val storeAddress = sc.parallelize(Seq(
       (Store("Ritual"), "1026 Valencia St"), (Store("Philz"), "748 Van Ne
       (Store("Philz"),"3101 24th St"), (Store("Starbucks"), "Seattle")))
val storeRating = sc.parallelize(Seq(
       (Store("Ritual"), 4.9), (Store("Philz"), 4.8)))
storeAddress.join(storeRating)
```

有时，我们不希望结果中的键必须在两个 RDD 中都存在。例如，在连接客户信息与推荐时，如果一些客户还没有收到推荐，我们仍然不希望丢掉这些顾客。`leftOuterJoin(other)` 和 `rightOuterJoin(other)` 都会根据键连接两个 RDD，但是允许结果中存在其中的一个 pair RDD 所缺失的键。

在使用 `leftOuterJoin()` 产生的 pair RDD 中，源 RDD 的每一个键都有对应的记录。每个键相应的值是由一个源 RDD 中的值与一个包含第二个 RDD 的值的 `Option`（在 Java 中为 `Optional`）对象组成的二元组。在 Python 中，如果一个值不存在，则使用 `None` 来表示；而数据存在时就用常规的值来表示，不使用任何封装。和 `join()` 一样，每个键可以得到多条记录；当这种情况发生时，我们会得到两个 RDD 中对应同一个键的两组值的笛卡尔积。


`Optional` 是 Google 的 Guava 库（https://github.com/google/guava）中的一部分，表示有可能缺失的值。可以调用 `isPresent()` 来看值是否存在，如果数据存在，则可以调用 `get()` 来获得其中包含的对象实例。

`rightOuterJoin()` 几乎与 `leftOuterJoin()` 完全一样，只不过预期结果中的键必须出现在第二个 RDD 中，而二元组中的可缺失的部分则来自于源 RDD 而非第二个 RDD。

回顾一下例 4-17，并在例 4-18 中对这两个之前用来演示 `join()` 的 pair RDD 进行 `leftOuterJoin()` 和 `rightOuterJoin()`。

例 4-18：`leftOuterJoin()` 与 `rightOuterJoin()`

```
storeAddress.leftOuterJoin(storeRating) ==
{(Store("Ritual"),("1026 Valencia St",Some(4.9))),
  (Store("Starbucks"),("Seattle",None)),
  (Store("Philz"),("748 Van Ness Ave",Some(4.8))),
  (Store("Philz"),("3101 24th St",Some(4.8)))}

storeAddress.rightOuterJoin(storeRating) ==
{(Store("Ritual"),(Some("1026 Valencia St"),4.9)),
  (Store("Philz"),(Some("748 Van Ness Ave"),4.8)),
  (Store("Philz"), (Some("3101 24th St"),4.8))}
```

4.3.4　数据排序

很多时候，让数据排好序是很有用的，尤其是在生成下游输出时。如果键有已定义的顺序，就可以对这种键值对 RDD 进行排序。当把数据排好序后，后续对数据进行 `collect()` 或 `save()` 等操作都会得到有序的数据。

我们经常要将 RDD 倒序排列，因此 `sortByKey()` 函数接收一个叫作 `ascending` 的参数，表示我们是否想要让结果按升序排序（默认值为 `true`）。有时我们也可能想按完全不同的排序依据进行排序。要支持这种情况，我们可以提供自定义的比较函数。例 4-19 至例 4-21 会将整数转为字符串，然后使用字符串比较函数来对 RDD 进行排序。

例 **4-19**：在 Python 中以字符串顺序对整数进行自定义排序

```
rdd.sortByKey(ascending=True, numPartitions=None, keyfunc = lambda x: str
```

例 **4-20**：在 Scala 中以字符串顺序对整数进行自定义排序

```
val input: RDD[(Int, Venue)] = ...
implicit val sortIntegersByString = new Ordering[Int] {
  override def compare(a: Int, b: Int) = a.toString.compare(b.toString)
}
rdd.sortByKey()
```

例 **4-21**：在 Java 中以字符串顺序对整数进行自定义排序

```
class IntegerComparator implements Comparator<Integer> {
   public int compare(Integer a, Integer b) {
     return String.valueOf(a).compareTo(String.valueOf(b))
   }
}
rdd.sortByKey(comp)
```

4.4 Pair RDD的行动操作

和转化操作一样，所有基础 RDD 支持的传统行动操作也都在 pair RDD 上可用。Pair RDD 提供了一些额外的行动操作，可以让我们充

分利用数据的键值对特性。这些操作列在了表 4-3 中。

表4-3：Pair RDD的行动操作（以键值对集合`{(1, 2), (3, 4), (3, 6)}`为例）

countByKey()	对每个键对应的元素分别计数
collectAsMap()	将结果以映射表的形式返回，以便查询
lookup(key)	返回给定键对应的所有值

就 pair RDD 而言，还有别的一些行动操作可以保存 RDD，会在第 5 章介绍。

4.5　数据分区（进阶）

本章要讨论的最后一个 Spark 特性是对数据集在节点间的分区进行控制。在分布式程序中，通信的代价是很大的，因此控制数据分布以获得最少的网络传输可以极大地提升整体性能。和单节点的程序需要为记录集合选择合适的数据结构一样，Spark 程序可以通过控制 RDD 分区方式来减少通信开销。分区并不是对所有应用都有好处的——比如，如果给定 RDD 只需要被扫描一次，我们完全没有必要对其预先进行分区处理。只有当数据集多次在诸如连接这种基于键的操作中使用时，分区才会有帮助。我们会给出一些小例子来说明这一点。

Spark 中所有的键值对 RDD 都可以进行分区。系统会根据一个针对键的函数对元素进行分组。尽管 Spark 没有给出显示控制每个键具体落在哪一个工作节点上的方法（部分原因是 Spark 即使在某些节点失败时依然可以工作），但 Spark 可以确保同一组的键出现在同一个节点上。比如，你可能使用哈希分区将一个 RDD 分成了 100 个分区，此时键的哈希值对 100 取模的结果相同的记录会被放在一个节点上。你也可以使用范围分区法，将键在同一个范围区间内的记录都放在同一个节点上。

举个简单的例子，我们分析这样一个应用，它在内存中保存着一张很大的用户信息表——也就是一个由 `(UserID, UserInfo)` 对组成的 RDD，其中 `UserInfo` 包含一个该用户所订阅的主题的列表。该应用会周期性地将这张表与一个小文件进行组合，这个小文件中存着过去五分钟内发生的事件——其实就是一个由 `(UserID, LinkInfo)` 对组成的表，存放着过去五分钟内某网站各用户的访问

情况。例如，我们可能需要对用户访问其未订阅主题的页面的情况进行统计。我们可以使用 Spark 的 `join()` 操作来实现这个组合操作，其中需要把 `UserInfo` 和 `LinkInfo` 的有序对根据 `UserID` 进行分组。我们的应用如例 4-22 所示。

例 **4-22**：简单的 Scala 应用

```
// 初始化代码；从HDFS上的一个Hadoop SequenceFile中读取用户信息
// userData中的元素会根据它们被读取时的来源，即HDFS块所在的节点来分布
// Spark此时无法获知某个特定的UserID对应的记录位于哪个节点上
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

// 周期性调用函数来处理过去五分钟产生的事件日志
// 假设这是一个包含(UserID, LinkInfo)对的SequenceFile
def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
  val joined = userData.join(events)// RDD of (UserID, (UserInfo, LinkInf
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Expand the tuple into its co
      !userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

这段代码可以正确运行，但是不够高效。这是因为在每次调用 `processNewLogs()` 时都会用到 `join()` 操作，而我们对数据集是如何分区的却一无所知。默认情况下，连接操作会将两个数据集中的所有键的哈希值都求出来，将该哈希值相同的记录通过网络传到同一台机器上，然后在那台机器上对所有键相同的记录进行连接操作（见图 4-4）。因为 `userData` 表比每五分钟出现的访问日志表 `events` 要大得多，所以要浪费时间做很多额外工作：在每次调用时都对 `userData` 表进行哈希值计算和跨节点数据混洗，虽然这些数据从来都不会变化。

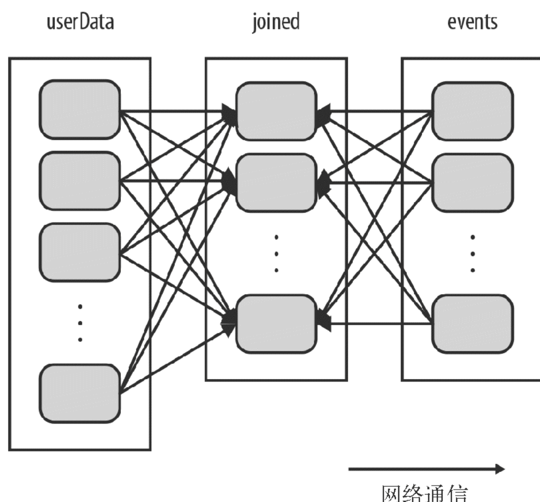


图 4-4：未使用 **partitionBy()** 时对 **userData** 和 **events** 进行连接操作

要解决这一问题也很简单：在程序开始时，对 `userData` 表使用 `partitionBy()` 转化操作，将这张表转为哈希分区。可以通过向 `partitionBy` 传递一个 `spark.HashPartitioner` 对象来实现该操作，如例 4-23 所示。

例 4-23：Scala 自定义分区方式

```
val sc = new SparkContext(...)
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
                 .partitionBy(new HashPartitioner(100))   // 构造100个分区
                 .persist()
```

`processNewLogs()` 方法可以保持不变：在 `processNewLogs()` 中，`events`RDD 是本地变量，只在该方法中使用了一次，所以为 `events` 指定分区方式没有什么用处。由于在构建 `userData` 时调用了 `partitionBy()`，Spark 就知道了该 RDD 是根据键的哈希值来分区的，这样在调用 `join()` 时，Spark 就会利用到这一点。具体来说，当调用 `userData.join(events)` 时，Spark 只会对 `events` 进行数据混洗操作，将 `events` 中特定 `UserID` 的记录发送到 `userData` 的对应分区所在的那台机器上（见图 4-5）。这样，需要通过网络传输的数据就大大减少了，程序运行速度也可以显著提升了。

注意，partitionBy() 是一个转化操作，因此它的返回值总是一个新的 RDD，但它不会改变原来的 RDD。RDD 一旦创建就无法修改。因此应该对 partitionBy() 的结果进行持久化，并保存为 userData，而不是原来的 sequenceFile() 的输出。此外，传给 partitionBy() 的 100 表示分区数目，它会控制之后对这个 RDD 进行进一步操作（比如连接操作）时有多少任务会并行执行。总的来说，这个值至少应该和集群中的总核心数一样。

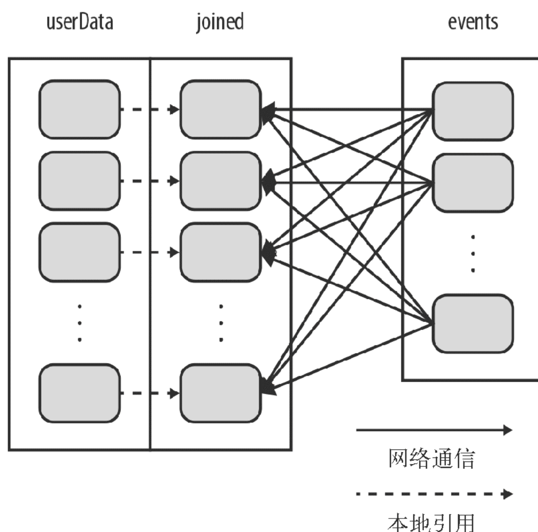


图 4-5：使用 **partitionBy()** 时对 **userData** 和 **events** 进行连接操作

> 如果没有将 partitionBy() 转化操作的结果持久化，那么后面每次用到这个 RDD 时都会重复地对数据进行分区操作。不进行持久化会导致整个 RDD 谱系图重新求值。那样的话，partitionBy() 带来的好处就会被抵消，导致重复对数据进行分区以及跨节点的混洗，和没有指定分区方式时发生的情况十分相似。

事实上，许多其他 Spark 操作会自动为结果RDD设定已知的分区方式信息，而且除 join() 外还有很多操作也会利用到已有的分区信息。比如，sortByKey() 和 groupByKey() 会分别生成范围分区的 RDD 和哈希分区的 RDD。而另一方面，诸如 map() 这样的操作会导致新的 RDD 失去父 RDD 的分区信息，因为这样的操作理论上可能会修改每条记录的键。接下来的几节中，我们会讨论如何获取 RDD

的分区信息，以及数据分区是如何影响各种 Spark 操作的。



Java 和 Python 中的数据分区

Spark 的 Java 和 Python 的 API 都和 Scala 的一样，可以从数据分区中获益。不过，在 Python 中，你不能将 `HashPartitioner` 对象传给 `partitionBy`，而只需要把需要的分区数传递过去（例如 `rdd.partitionBy(100)`）。

4.5.1　获取RDD的分区方式

在 Scala 和 Java 中，你可以使用 RDD 的 `partitioner` 属性（Java 中使用 `partitioner()` 方法）来获取 RDD 的分区方式。[5] 它会返回一个 `scala.Option` 对象，这是 Scala 中用来存放可能存在的对象的容器类。你可以对这个 `Option` 对象调用 `isDefined()` 来检查其中是否有值，调用 `get()` 来获取其中的值。如果存在值的话，这个值会是一个 `spark.Partitioner` 对象。这本质上是一个告诉我们 RDD 中各个键分别属于哪个分区的函数。我们之后会进一步讨论这一点。

> 5Python API 没有提供查询分区方式的方法，但是 Spark 内部仍然会利用已有的分区信息。

在 Spark shell 中使用 `partitioner` 属性不仅是检验各种 Spark 操作如何影响分区方式的一种好办法，还可以用来在你的程序中检查想要使用的操作是否会生成正确的结果（见例 4-24）。

例 **4-24**：获取 RDD 的分区方式

```
import org.apache.spark
scala> val pairs = sc.parallelize(List((1, 1), (2, 2), (3, 3)))
pairs: spark.RDD[(Int, Int)] = ParallelCollectionRDD[0] at parallelize at
<console>:12

scala> pairs.partitioner
res0: Option[spark.Partitioner] = None

scala> val partitioned = pairs.partitionBy(new spark.HashPartitioner(2))
partitioned: spark.RDD[(Int, Int)] = ShuffledRDD[1] at partitionBy at <con

scala> partitioned.partitioner
```

```
res1: Option[spark.Partitioner] = Some(spark.HashPartitioner@5147788d)
```

在这段简短的代码中，我们创建出了一个由 (Int, Int) 对组成的 RDD，初始时没有分区方式信息（一个值为 None 的 Option 对象）。然后通过对第一个 RDD 进行哈希分区，创建出了第二个 RDD。如果确实要在后续操作中使用 partitioned，那就应当在定义 partitioned 时，在第三行输入的最后加上 persist()。这和之前的例子中需要对 userData 调用 persist() 的原因是一样的：如果不调用 persist() 的话，后续的 RDD 操作会对 partitioned 的整个谱系重新求值，这会导致对 pairs 一遍又一遍地进行哈希分区操作。

4.5.2　从分区中获益的操作

Spark 的许多操作都引入了将数据根据键跨节点进行混洗的过程。所有这些操作都会从数据分区中获益。就 Spark 1.0 而言，能够从数据分区中获益的操作有 cogroup()、groupWith()、join()、leftOuterJoin()、rightOuterJoin()、groupByKey()、reduceByKey()、combineByKey() 以及 lookup()。

对于像 reduceByKey() 这样只作用于单个 RDD 的操作，运行在未分区的 RDD 上的时候会导致每个键的所有对应值都在每台机器上进行本地计算，只需要把本地最终归约出的结果值从各工作节点传回主节点，所以原本的网络开销就不算大。而对于诸如 cogroup() 和 join() 这样的二元操作，预先进行数据分区会导致其中至少一个 RDD（使用已知分区器的那个 RDD）不发生数据混洗。如果两个 RDD 使用同样的分区方式，并且它们还缓存在同样的机器上（比如一个 RDD 是通过 mapValues() 从另一个 RDD 中创建出来的，这两个 RDD 就会拥有相同的键和分区方式），或者其中一个 RDD 还没有被计算出来，那么跨节点的数据混洗就不会发生了。

4.5.3　影响分区方式的操作

Spark 内部知道各操作会如何影响分区方式，并将会对数据进行分区的操作的结果 RDD 自动设置为对应的分区器。例如，如果你调用 join() 来连接两个 RDD；由于键相同的元素会被哈希到同一台机器上，Spark 知道输出结果也是哈希分区的，这样对连接的结果进行诸如 reduceByKey() 这样的操作时就会明显变快。

不过，转化操作的结果并不一定会按已知的分区方式分区，这时输出

的 RDD 可能就会没有设置分区器。例如，当你对一个哈希分区的键值对 RDD 调用 map() 时，由于传给 map() 的函数理论上可以改变元素的键，因此结果就不会有固定的分区方式。Spark 不会分析你的函数来判断键是否会被保留下来。不过，Spark 提供了另外两个操作 mapValues() 和 flatMapValues() 作为替代方法，它们可以保证每个二元组的键保持不变。

这里列出了所有会为生成的结果 RDD 设好分区方式的操作：cogroup()、groupWith()、join()、leftOuterJoin()、rightOuterJoin()、groupByKey()、reduceByKey()、combineByKey()、partitionBy()、sort()、mapValues()（如果父 RDD 有分区方式的话）、flatMapValues()（如果父 RDD 有分区方式的话），以及 filter()（如果父 RDD 有分区方式的话）。其他所有的操作生成的结果都不会存在特定的分区方式。

最后，对于二元操作，输出数据的分区方式取决于父 RDD 的分区方式。默认情况下，结果会采用哈希分区，分区的数量和操作的并行度一样。不过，如果其中的一个父 RDD 已经设置过分区方式，那么结果就会采用那种分区方式；如果两个父 RDD 都设置过分区方式，结果 RDD 会采用第一个父 RDD 的分区方式。

4.5.4　示例：PageRank

PageRank 是一种从 RDD 分区中获益的更复杂的算法，我们以它为例进行分析。PageRank 算法是以 Google 的拉里· 佩吉（Larry Page）的名字命名的，用来根据外部文档指向一个文档的链接，对集合中每个文档的重要程度赋一个度量值。该算法可以用于对网页进行排序，当然，也可以用于排序科技文章或社交网络中有影响的用户。

PageRank 是执行多次连接的一个迭代算法，因此它是 RDD 分区操作的一个很好的用例。算法会维护两个数据集：一个由 (pageID, linkList) 的元素组成，包含每个页面的相邻页面的列表；另一个由 (pageID, rank) 元素组成，包含每个页面的当前排序值。它按如下步骤进行计算。

(1) 将每个页面的排序值初始化为 1.0。

(2) 在每次迭代中，对页面 p，向其每个相邻页面（有直接链接的页面）发送一个值为 rank(p)/numNeighbors(p) 的贡献值。

(3) 将每个页面的排序值设为 0.15 + 0.85 *

contributionsReceived。

最后两步会重复几个循环，在此过程中，算法会逐渐收敛于每个页面的实际 PageRank 值。在实际操作中，收敛通常需要大约 10 轮迭代。

例 4-25 给出了使用 Spark 实现 PageRank 的代码。

例 **4-25**：Scala 版 PageRank

```
// 假设相邻页面列表以Spark objectFile的形式存储
val links = sc.objectFile[(String, Seq[String])]("links")
              .partitionBy(new HashPartitioner(100))
              .persist()

// 将每个页面的排序值初始化为1.0；由于使用mapValues，生成的RDD
// 的分区方式会和"links"的一样
var ranks = links.mapValues(v => 1.0)

// 运行10轮PageRank迭代
for(i <- 0 until 10) {
  val contributions = links.join(ranks).flatMap {
    case (pageId, (links, rank)) =>
      links.map(dest => (dest, rank / links.size))
  }
  ranks = contributions.reduceByKey((x, y) => x + y).mapValues(v => 0.15 
}

// 写出最终排名
ranks.saveAsTextFile("ranks")
```

这就行了！算法从将 ranksRDD 的每个元素的值初始化为 1.0 开始，然后在每次迭代中不断更新 ranks 变量。在 Spark 中编写 PageRank 的主体相当简单：首先对当前的 ranksRDD 和静态的 linksRDD 进行一次 join() 操作，来获取每个页面 ID 对应的相邻页面列表和当前的排序值，然后使用 flatMap 创建出“contributions”来记录每个页面对各相邻页面的贡献。然后再把这些贡献值按照页面 ID（根据获得共享的页面）分别累加起来，把该页面的排序值设为 0.15 + 0.85 * contributionsReceived。

虽然代码本身很简单，这个示例程序还是做了不少事情来确保 RDD 以比较高效的方式进行分区，以最小化通信开销：

(1) 请注意，linksRDD 在每次迭代中都会和 ranks 发生连接操作。由于 links 是一个静态数据集，所以我们在程序一开始的时候

就对它进行了分区操作，这样就不需要把它通过网络进行数据混洗了。实际上，linksRDD 的字节数一般来说也会比 ranks 大很多，毕竟它包含每个页面的相邻页面列表（由页面 ID 组成），而不仅仅是一个 Double 值，因此这一优化相比 PageRank 的原始实现（例如普通的 MapReduce）节约了相当可观的网络通信开销。

(2) 出于同样的原因，我们调用 links 的 persist() 方法，将它保留在内存中以供每次迭代使用。

(3) 当我们第一次创建 ranks 时，我们使用 mapValues() 而不是 map() 来保留父 RDD（links）的分区方式，这样对它进行的第一次连接操作就会开销很小。

(4) 在循环体中，我们在 reduceByKey() 后使用 mapValues()；因为 reduceByKey() 的结果已经是哈希分区的了，这样一来，下一次循环中将映射操作的结果再次与 links 进行连接操作时就会更加高效。



为了最大化分区相关优化的潜在作用，你应该在无需改变元素的键时尽量使用 mapValues() 或 flatMapValues()。

4.5.5　自定义分区方式

虽然 Spark 提供的 HashPartitioner 与 RangePartitioner 已经能够满足大多数用例，但 Spark 还是允许你通过提供一个自定义的 Partitioner 对象来控制 RDD 的分区方式。这可以让你利用领域知识进一步减少通信开销。

举个例子，假设我们要在一个网页的集合上运行前一节中的 PageRank 算法。在这里，每个页面的 ID（RDD 中的键）是页面的 URL。当我们使用简单的哈希函数进行分区时，拥有相似的 URL 的页面（比如 http://www.cnn.com/WORLD 和 http://www.cnn.com/US）可能会被分到完全不同的节点上。然而，我们知道在同一个域名下的网页更有可能相互链接。由于 PageRank 需要在每次迭代中从每个页面向它所有相邻的页面发送一条消息，因此把这些页面分组到同一个分区中会更好。可以使用自定义的分区器来实现仅根据域名而不是整个 URL 来分区。

要实现自定义的分区器，你需要继承 org.apache.spark.Partitioner 类并实现下面三个方法。

- `numPartitions: Int`：返回创建出来的分区数。
- `getPartition(key: Any): Int`：返回给定键的分区编号（0 到 `numPartitions-1`）。
- `equals()`：Java 判断相等性的标准方法。这个方法的实现非常重要，Spark 需要用这个方法来检查你的分区器对象是否和其他分区器实例相同，这样 Spark 才可以判断两个 RDD 的分区方式是否相同。

有一个问题需要注意，当你的算法依赖于 Java 的 `hashCode()` 方法时，这个方法有可能会返回负数。你需要十分谨慎，确保 `getPartition()` 永远返回一个非负数。

例 4-26 展示了如何编写一个前面构思的基于域名的分区器，这个分区器只对 URL 中的域名部分求哈希。

例 **4-26**：Scala 自定义分区方式

```
class DomainNamePartitioner(numParts: Int) extends Partitioner {
  override def numPartitions: Int = numParts
  override def getPartition(key: Any): Int = {
    val domain = new Java.net.URL(key.toString).getHost()
    val code = (domain.hashCode % numPartitions)
    if(code < 0) {
      code + numPartitions // 使其非负
    }else{
      code
    }
  }
  // 用来让Spark区分分区函数对象的Java equals方法
  override def equals(other: Any): Boolean = other match {
    case dnp: DomainNamePartitioner =>
      dnp.numPartitions == numPartitions
    case _ =>
      false
  }
}
```

注意，在 `equals()` 方法中，使用 Scala 的模式匹配操作符（`match`）来检查 `other` 是否是 `DomainNamePartitioner`，并在成立时自动进行类型转换；这和 Java 中的 `instanceof()` 是一样的。

使用自定义的 `Partitioner` 是很容易的：只要把它传给

partitionBy() 方法即可。Spark 中有许多依赖于数据混洗的方法，比如 join() 和 groupByKey()，它们也可以接收一个可选的 Partitioner 对象来控制输出数据的分区方式。

在 Java 中创建一个自定义 Partitioner 的方法与 Scala 中的做法非常相似：只需要扩展 spark.Partitioner 类并且实现必要的方法即可。

在 Python 中，不需要扩展 Partitioner 类，而是把一个特定的哈希函数作为一个额外的参数传给 RDD.partitionBy() 函数，如例 4-27 所示。

例 **4-27**：Python 自定义分区方式

```
import urlparse

def hash_domain(url):
  return hash(urlparse.urlparse(url).netloc)

rdd.partitionBy(20, hash_domain)  # 创建20个分区
```

注意，这里你所传过去的哈希函数会被与其他 RDD 的分区函数区分开来。如果你想要对多个 RDD 使用相同的分区方式，就应该使用同一个函数对象，比如一个全局函数，而不是为每个 RDD 创建一个新的函数对象。

4.6　总结

本章我们学习了如何使用 Spark 提供的专门的函数来操作键值对数据。第 3 章中讲到的技巧也同样适用于 pair RDD。在下一章中我们会介绍如何读取和保存数据。

第 5 章　数据读取与保存

本章对于工程师和数据科学家都较为实用。工程师会了解到更多的输出格式，有利于找到非常适合用于下游处理程序的格式。数据科学家则可能更关心数据的现有的组织形式。

5.1　动机

我们已经学了很多在 Spark 中对已分发的数据执行的操作。到目前为止，所展示的示例都是从本地集合或者普通文件中进行数据读取和保存的。但有时候，数据量可能大到无法放在一台机器中，这时就需要探索别的数据读取和保存的方法了。

Spark 支持很多种输入输出源。一部分原因是 Spark 本身是基于 Hadoop 生态圈而构建，特别是 Spark 可以通过 Hadoop MapReduce 所使用的 `InputFormat` 和 `OutputFormat` 接口访问数据，而大部分常见的文件格式与存储系统（例如 S3、HDFS、Cassandra、HBase 等）都支持这种接口。[1] 5.2.6 节展示了如何直接使用这些格式。

> 1`InputFormat` 和 `OutputFormat` 是 MapReduce 中用来连接数据源的 Java API。

不过，基于这些原始接口构建出的高层 API 会更常用。幸运的是，Spark 及其生态系统提供了很多可选方案。本章会介绍以下三类常见的数据源。

- 文件格式与文件系统

  对于存储在本地文件系统或分布式文件系统（比如 NFS、HDFS、Amazon S3 等）中的数据，Spark 可以访问很多种不同的文件格式，包括文本文件、JSON、SequenceFile，以及 protocol buffer。我们会展示几种常见格式的用法，以及 Spark 针对不同文件系统的配置和压缩选项。

- **Spark SQL**中的结构化数据源

  第 9 章会介绍 Spark SQL 模块，它针对包括 JSON 和 Apache Hive 在内的结构化数据源，为我们提供了一套更加简洁高效的

API。此处会粗略地介绍一下如何使用 Spark SQL，而大部分细节将留到第 9 章讲解。

- 数据库与键值存储

  本章还会概述 Spark 自带的库和一些第三方库，它们可以用来连接 Cassandra、HBase、Elasticsearch 以及 JDBC 源。

这里选择的大多数方法都支持 Spark 所支持的三种编程语言，但是还是有一些库只支持 Java 和 Scala。这种情况我们会专门指出。

5.2　文件格式

Spark 对很多种文件格式的读取和保存方式都很简单。从诸如文本文件的非结构化的文件，到诸如 JSON 格式的半结构化的文件，再到诸如 SequenceFile 这样的结构化的文件，Spark 都可以支持（见表 5-1）。Spark 会根据文件扩展名选择对应的处理方式。这一过程是封装好的，对用户透明。

表**5-1**：**Spark**支持的一些常见格式

格式名称	结构化	备注
文本文件	否	普通的文本文件，每行一条记录
JSON	半结构化	常见的基于文本的格式，半结构化；大多数库都要求每行一条记录
CSV	是	非常常见的基于文本的格式，通常在电子表格应用中使用
SequenceFiles	是	一种用于键值对数据的常见 Hadoop 文件格式
Protocol buffers	是	一种快速、节约空间的跨语言格式
对象文件	是	用来将 Spark 作业中的数据存储下来以让共享的代码读取。改变类的时候它会失效，因为它依赖于 Java 序列化

除了 Spark 中直接支持的输出机制，还可以对键数据（或成对数据）使用 Hadoop 的新旧文件 API。由于 Hadoop 接口要求使用键值对数据，所以也只能这样用，即使有些格式事实上忽略了键。对于那些会忽视键的格式，通常使用假的键（比如 `null`）。

5.2.1　文本文件

在 Spark 中读写文本文件很容易。当我们将一个文本文件读取为 RDD 时，输入的每一行都会成为 RDD 的一个元素。也可以将多个完整的文本文件一次性读取为一个 pair RDD，其中键是文件名，值是文件内容。

1. 读取文本文件

只需要使用文件路径作为参数调用 SparkContext 中的 `textFile()` 函数，就可以读取一个文本文件，如例 5-1 至例 5-3 所示。如果要控制分区数的话，可以指定 `minPartitions`。

例 **5-1**：在 Python 中读取一个文本文件

```
input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

例 **5-2**：在 Scala 中读取一个文本文件

```
val input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

例 **5-3**：在 Java 中读取一个文本文件

```
JavaRDD<String> input = sc.textFile("file:///home/holden/repos/spark/READM
```

如果多个输入文件以一个包含数据所有部分的目录的形式出现，可以用两种方式来处理。可以仍使用 `textFile` 函数，传递目录作为参数，这样它会把各部分都读取到 RDD 中。有时候有必要知道数据的各部分分别来自哪个文件（比如将键放在文件名中的时间数据），有时候则希望同时处理整个文件。如果文件足够小，那么可以使用 `SparkContext.wholeTextFiles()` 方法，该方法会返回一个 pair RDD，其中键是输入文件的文件名。

`wholeTextFiles()` 在每个文件表示一个特定时间段内的数据时非常有用。如果有表示不同阶段销售数据的文件，则可以很容易地求出每个阶段的平均值，如例 5-4 所示。

例 **5-4**：在 Scala 中求每个文件的平均值

```
val input = sc.wholeTextFiles("file://home/holden/salesFiles")
val result = input.mapValues{y =>
  val nums = y.split(" ").map(x => x.toDouble)
  nums.sum / nums.size.toDouble
}
```



Spark 支持读取给定目录中的所有文件，以及在

输入路径中使用通配字符（如 `part-*.txt`）。大规模数据集通常存放在多个文件中，因此这一特性很有用，尤其是在同一目录中存在一些别的文件（比如成功标记文件）的时候。

2. 保存文本文件

输出文本文件也相当简单。例 5-5 中演示的 `saveAsTextFile()` 方法接收一个路径，并将 RDD 中的内容都输入到路径对应的文件中。Spark 将传入的路径作为目录对待，会在那个目录下输出多个文件。这样，Spark 就可以从多个节点上并行输出了。在这个方法中，我们不能控制数据的哪一部分输出到哪个文件中，不过有些输出格式支持控制。

例 **5-5**：在 Python 中将数据保存为文本文件

```
result.saveAsTextFile(outputFile)
```

5.2.2　JSON

JSON 是一种使用较广的半结构化数据格式。读取 JSON 数据的最简单的方式是将数据作为文本文件读取，然后使用 JSON 解析器来对 RDD 中的值进行映射操作。类似地，也可以使用我们喜欢的 JSON 序列化库来将数据转为字符串，然后将其写出去。在 Java 和 Scala 中也可以使用一个自定义 Hadoop 格式来操作 JSON 数据。9.3.3 节还会展示如何使用 Spark SQL 读取 JSON 数据。

1. 读取JSON

将数据作为文本文件读取，然后对 JSON 数据进行解析，这样的方法可以在所有支持的编程语言中使用。这种方法假设文件中的每一行都是一条 JSON 记录。如果你有跨行的 JSON 数据，你就只能读入整个文件，然后对每个文件进行解析。如果在你使用的语言中构建一个 JSON 解析器的开销较大，你可以使用 `mapPartitions()` 来重用解析器。请参考 6.4 节了解详情。

我们使用的这三种编程语言中有大量可用的 JSON 库，为了简单起见，这里只为每种语言介绍一种库。Python 中使用的是内建的库（https://docs.python.org/2/library/json.html，见例 5-6），而在 Java 和 Scala 中则会使用 Jackson（http://jackson.codehaus.org/，

见例 5-7 和例 5-8）。之所以选择这些库，是因为它们性能还不错，而且使用起来比较简单。如果你 在解析阶段花费了大量的时间，你就应该选择 Scala（http://engineering.ooyala.com/blog/comparing-scala-json-libraries）或 Java（http://geokoder.com/java-json-libraries-comparison）中别的 JSON 库。

例 **5-6**：在 Python 中读取非结构化的 JSON

```
import json
data = input.map(lambda x: json.loads(x))
```

在 Scala 和 Java 中，通常将记录读入到一个代表结构信息的类中。在这个过程中可能还需要略过一些无效的记录。下面以将记录读取为 Person 类作为一个例子。

例 **5-7**：在 Scala 中读取 JSON

```
import com.fasterxml.jackson.module.scala.DefaultScalaModule
import com.fasterxml.jackson.module.scala.experimental.ScalaObjectMapper
import com.fasterxml.jackson.databind.ObjectMapper
import com.fasterxml.jackson.databind.DeserializationFeature
...
case class Person(name: String, lovesPandas: Boolean) // 必须是顶级类
...
// 将其解析为特定的case class。使用flatMap，通过在遇到问题时返回空列表（None）
// 来处理错误，而在没有问题时返回包含一个元素的列表（Some(_)）
val result = input.flatMap(record => {
  try {
    Some(mapper.readValue(record, classOf[Person]))
  } catch {
    case e: Exception => None
  }})
```

例 **5-8**：在 Java 中读取 JSON

```
class ParseJson implements FlatMapFunction<Iterator<String>, Person> {
  public Iterable<Person> call(Iterator<String> lines) throws Exception {
    ArrayList<Person> people = new ArrayList<Person>();
    ObjectMapper mapper = new ObjectMapper();
    while (lines.hasNext()) {
      String line = lines.next();
      try {
        people.add(mapper.readValue(line, Person.class));
      } catch (Exception e) {
        // 跳过失败的数据
```

```
      }
    }
    return people;
  }
}
JavaRDD<String> input = sc.textFile("file.json");
JavaRDD<Person> result = input.mapPartitions(new ParseJson());
```



处理格式不正确的记录有可能会引起很严重的问题，尤其对于像 JSON 这样的半结构化数据来说。对于小数据集来说，可以接受在遇到错误的输入时停止程序（程序失败），但是对于大规模数据集来说，格式错误是家常便饭。如果选择跳过格式不正确的数据，你应该尝试使用累加器来跟踪错误的个数。

2. 保存JSON

写出 JSON 文件比读取它要简单得多，因为不需要考虑格式错误的数据，并且也知道要写出的数据的类型。可以使用之前将字符串 RDD 转为解析好的 JSON 数据的库，将由结构化数据组成的 RDD 转为字符串 RDD，然后使用 Spark 的文本文件 API 写出去。

假设我们要选出喜爱熊猫的人，就可以从第一步中获取输入数据，然后筛选出喜爱熊猫的人，如例 5-9 至例 5-11 所示。

例 **5-9**：在 Python 保存为 JSON

```
(data.filter(lambda x: x["lovesPandas"]).map(lambda x: json.dumps(x))
  .saveAsTextFile(outputFile))
```

例 **5-10**：在 Scala 中保存为 JSON

```
result.filter(p => P.lovesPandas).map(mapper.writeValueAsString(_))
  .saveAsTextFile(outputFile)
```

例 **5-11**：在 Java 中保存为 JSON

```
class WriteJson implements FlatMapFunction<Iterator<Person>, String> {
  public Iterable<String> call(Iterator<Person> people) throws Exception {
    ArrayList<String> text = new ArrayList<String>();
    ObjectMapper mapper = new ObjectMapper();
    while (people.hasNext()) {
```

```
            Person person = people.next();
            text.add(mapper.writeValueAsString(person));
        }
        return text;
    }
}

JavaRDD<Person> result = input.mapPartitions(new ParseJson()).filter(
    new LikesPandas());
JavaRDD<String> formatted = result.mapPartitions(new WriteJson());
formatted.saveAsTextFile(outfile);
```

这样一来，就可以通过已有的操作文本数据的机制和 JSON 库，使用 Spark 轻易地读取和保存 JSON 数据了。

5.2.3　逗号分隔值与制表符分隔值

逗号分隔值（CSV）文件每行都有固定数目的字段，字段间用逗号隔开（在制表符分隔值文件，即 TSV 文件中用制表符隔开）。记录通常是一行一条，不过也不总是这样，有时也可以跨行。CSV 文件和 TSV 文件有时支持的标准并不一致，主要是在处理换行符、转义字符、非 ASCII 字符、非整数值等方面。CSV 原生并不支持嵌套字段，所以需要手动组合和分解特定的字段。

与 JSON 中的字段不一样的是，这里的每条记录都没有相关联的字段名，只能得到对应的序号。常规做法是使用第一行中每列的值作为字段名。

1. 读取CSV

读取 CSV/TSV 数据和读取 JSON 数据相似，都需要先把文件当作普通文本文件来读取数据，再对数据进行处理。由于格式标准的缺失，同一个库的不同版本有时也会用不同的方式处理输入数据。

与 JSON 一样，CSV 也有很多不同的库，但是只在每种语言中使用一个库。同样，对于 Python 我们会使用自带的 csv 库（https://docs.python.org/2/library/csv.html）。在 Scala 和 Java 中则使用 opencsv 库（http://opencsv.sourceforge.net/）。



Hadoop InputFormat 中的 `CSVInputFormat`（http://docs.oracle.com/cd/E27101_01/appdev.10/e20858/oracle/hadoop/loader/examples/CSVInputFormat.html）也可以用

于在 Scala 和 Java 中读取 CSV 数据。不过它不支持包含换行符的记录。

如果恰好你的 CSV 的所有数据字段均没有包含换行符，你也可以使用 `textFile()` 读取并解析数据，如例 5-12 至例 5-14 所示。

例 5-12：在 Python 中使用 `textFile()` 读取 CSV

```
import csv
import StringIO
...
def loadRecord(line):
    """解析一行CSV记录"""
    input = StringIO.StringIO(line)
    reader = csv.DictReader(input, fieldnames=["name", "favouriteAnimal"])
    return reader.next()
input = sc.textFile(inputFile).map(loadRecord)
```

例 5-13：在 Scala 中使用 `textFile()` 读取 CSV

```
import Java.io.StringReader
import au.com.bytecode.opencsv.CSVReader
...
val input = sc.textFile(inputFile)
val result = input.map{ line =>
  val reader = new CSVReader(new StringReader(line));
  reader.readNext();
}
```

例 5-14：在 Java 中使用 `textFile()` 读取 CSV

```
import au.com.bytecode.opencsv.CSVReader;
import Java.io.StringReader;
...
public static class ParseLine implements Function<String, String[]> {
  public String[] call(String line) throws Exception {
    CSVReader reader = new CSVReader(new StringReader(line));
    return reader.readNext();
  }
}
JavaRDD<String> csvFile1 = sc.textFile(inputFile);
JavaPairRDD<String[]> csvData = csvFile1.map(new ParseLine());
```

如果在字段中嵌有换行符，就需要完整读入每个文件，然后解析各段，如例 5-15 至例 5-17 所示。如果每个文件都很大，读取和解析的过程可能会很不幸地成为性能瓶颈。读取文本文件的其他方法在 5.2.1 节中也有所提及。

例 **5-15**：在 Python 中完整读取 CSV

```
def loadRecords(fileNameContents):
    """读取给定文件中的所有记录"""
    input = StringIO.StringIO(fileNameContents[1])
    reader = csv.DictReader(input, fieldnames=["name", "favoriteAnimal"])
    return reader
fullFileData = sc.wholeTextFiles(inputFile).flatMap(loadRecords)
```

例 **5-16**：在 Scala 中完整读取 CSV

```
case class Person(name: String, favoriteAnimal: String)

val input = sc.wholeTextFiles(inputFile)
val result = input.flatMap{ case (_, txt) =>
  val reader = new CSVReader(new StringReader(txt));
  reader.readAll().map(x => Person(x(0), x(1)))
}
```

例 **5-17**：在 Java 中完整读取 CSV

```
public static class ParseLine
  implements FlatMapFunction<Tuple2<String, String>, String[]> {
  public Iterable<String[]> call(Tuple2<String, String> file) throws Excep
    CSVReader reader = new CSVReader(new StringReader(file._2()));
    return reader.readAll();
  }
}
JavaPairRDD<String, String> csvData = sc.wholeTextFiles(inputFile);
JavaRDD<String[]> keyedRDD = csvData.flatMap(new ParseLine());
```



如果只有一小部分输入文件，你需要使用 `wholeFile()` 方法，可能还需要对输入数据进行重新分区使得 Spark 能够更高效地并行化执行后续操作。

2. 保存CSV

和 JSON 数据一样，写出 CSV/TSV 数据相当简单，同样可以通过重用输出编码器来加速。由于在 CSV 中我们不会在每条记录中输出字段名，因此为了使输出保持一致，需要创建一种映射关系。一种简单做法是写一个函数，用于将各字段转为指定顺序的数组。在 Python 中，如果输出字典，CSV 输出器会根据创建输出器时给定的 `fieldnames` 的顺序帮我们完成这一行为。

我们所使用的 CSV 库要输出到文件或者输出器，所以可以使用 `StringWriter` 或 `StringIO` 来将结果放到 RDD 中，如例 5-18 和例 5-19 所示。

例 **5-18**：在 Python 中写 CSV

```
def writeRecords(records):
    """写出一些CSV记录"""
    output = StringIO.StringIO()
    writer = csv.DictWriter(output, fieldnames=["name", "favoriteAnimal"])
    for record in records:
        writer.writerow(record)
    return [output.getvalue()]

pandaLovers.mapPartitions(writeRecords).saveAsTextFile(outputFile)
```

例 **5-19**：在 Scala 中写 CSV

```
pandaLovers.map(person => List(person.name, person.favoriteAnimal).toArray
.mapPartitions{people =>
  val stringWriter = new StringWriter();
  val csvWriter = new CSVWriter(stringWriter);
  csvWriter.writeAll(people.toList)
  Iterator(stringWriter.toString)
}.saveAsTextFile(outFile)
```

你可能已经注意到，前面的例子只能在我们知道所要输出的所有字段时使用。然而，如果一些字段名是在运行时由用户输入决定的，就要使用别的方法了。最简单的方法是遍历所有的数据，提取不同的键，然后分别输出。

5.2.4　SequenceFile

SequenceFile 是由没有相对关系结构的键值对文件组成的常用 Hadoop 格式。SequenceFile 文件有同步标记，Spark可以用它来定位

到文件中的某个点，然后再与记录的边界对齐。这可以让 Spark 使用多个节点高效地并行读取 SequenceFile 文件。SequenceFile 也是 Hadoop MapReduce 作业中常用的输入输出格式，所以如果你在使用一个已有的 Hadoop 系统，数据很有可能是以 SequenceFile 的格式供你使用的。

由于 Hadoop 使用了一套自定义的序列化框架，因此 SequenceFile 是由实现 Hadoop 的 Writable 接口的元素组成。表 5-2 列出了一些常见的数据类型以及它们对应的 Writable 类。标准的经验法则是尝试在类名的后面加上 Writable 这个词，然后检查它是否是 `org.apache.hadoop.io.Writable`（http://hadoop.apache.org/docs/r2.4.1/api/org/apache/hadoop/io/Writable.html）已知的子类。如果你无法为要写出的数据找到对应的 Writable 类型（比如自定义的 case class），你可以通过重载 `org.apache.hadoop.io.Writable` 中的 `readfields` 和 `write` 来实现自己的 Writable 类。



Hadoop 的 RecordReader 会为每条记录重用同一个对象，因此直接调用 RDD 的 `cache` 会导致失败；实际上，你只需要使用一个简单的 `map()` 操作然后将结果缓存即可。还有，许多 Hadoop Writable 类没有实现 `java.io.Serializable` 接口，因此为了让它们能在 RDD 中使用，还是要用 `map()` 来转换它们。

表5-2：Hadoop Writable类型对应表

Scala类型	Java类型	Hadoop Writable类
Int	Integer	IntWritable 或 VIntWritable2
Long	Long	LongWritable 或 VLongWritable2
Float	Float	FloatWritable
Double	Double	DoubleWritable
Boolean	Boolean	BooleanWritable
Array[Byte]	byte[]	BytesWritable
String	String	Text
Array[T]	T[]	ArrayWritable<TW>3
List[T]	List<T>	ArrayWritable<TW>3
Map[A, B]	Map<A, B>	MapWritable<AW, BW>3

2整型和长整型通常存储为定长的形式。存储数字 12 占据的空间和存储数字 2**30 占据的一样。如果你有大量的小数据，你应该使用可变长的类型 `VIntWritable` 和 `VLongWritable`，它们可以在存储较小数值时使用更少的位。

3模板类型也必须使用 Writable 类型。

在 Spark 1.0 以及更早版本中，SequenceFile 只能在 Java 和 Scala 中使用，不过 Spark 1.1 加入了在 Python 中读取和保存 SequenceFile 的功能。但要注意，你还是需要使用 Java 或 Scala 来实现自定义 Writable 类。Spark 的 Python API 只能将 Hadoop 中存在的基本 Writable 类型转为 Python 类型，并尽量基于可用的 getter 方法处理别的类型。

1. 读取SequenceFile

Spark 有专门用来读取 SequenceFile 的接口。在 SparkContext 中，可以调用 `sequenceFile(path, keyClass, valueClass, minPartitions)`。前面提到过，SequenceFile 使用 Writable 类，因此 `keyClass` 和 `valueClass` 参数都必须使用正确的 Writable 类。举个例子，假设要从一个 SequenceFile 中读取人员以及他们所见过的熊猫数目。在这个例子中，`keyClass` 是 `Text`，而 `valueClass` 则是 `IntWritable` 或 `VIntWritable`。为了方便演示，在例 5-20 至例 5-22 中使用 `IntWritable`。

例 **5-20**：在 Python 读取 SequenceFile

```
data = sc.sequenceFile(inFile,
  "org.apache.hadoop.io.Text", "org.apache.hadoop.io.IntWritable")
```

例 **5-21**：在 Scala 中读取 SequenceFile

```
val data = sc.sequenceFile(inFile, classOf[Text], classOf[IntWritable]).
  map{case (x, y) => (x.toString, y.get())}
```

例 **5-22**：在 Java 中读取 SequenceFile

```
public static class ConvertToNativeTypes implements
  PairFunction<Tuple2<Text, IntWritable>, String, Integer> {
  public Tuple2<String, Integer> call(Tuple2<Text, IntWritable> record) {
    return new Tuple2(record._1.toString(), record._2.get());
  }
}

JavaPairRDD<Text, IntWritable> input = sc.sequenceFile(fileName, Text.clas
  IntWritable.class);
JavaPairRDD<String, Integer> result = input.mapToPair(
```

```
new ConvertToNativeTypes());
```

 在 Scala 中有一个很方便的函数可以自动将 Writable 对象转为相应的 Scala 类型。可以调用 `sequenceFile[Key, Value](path, minPartitions)` 返回 Scala 原生数据类型的 RDD，而无需指定 `keyClass` 和 `valueClass`。

2. 保存SequenceFile

在 Scala 中将数据写出到 SequenceFile 的做法也很类似。首先，因为 SequenceFile 存储的是键值对，所以需要创建一个由可以写出到 SequenceFile 的类型构成的 `PairRDD`。我们已经进行了将许多 Scala 的原生类型转为 Hadoop Writable 的隐式转换，所以如果你要写出的是 Scala 的原生类型，可以直接调用 `saveSequenceFile(path)` 保存你的 `PairRDD`，它会帮你写出数据。如果键和值不能自动转为 Writable 类型，或者想使用变长类型（比如 `VIntWritable`），就可以对数据进行映射操作，在保存之前进行类型转换。让我们改写之前的那个例子（人员以及他们所见过的熊猫数目），如例 5-23 所示。

例 **5-23**：在 Scala 中保存 SequenceFile

```
val data = sc.parallelize(List(("Panda", 3), ("Kay", 6), ("Snail", 2)))
data.saveAsSequenceFile(outputFile)
```

在 Java 中保存 SequenceFile 要稍微复杂一些，因为 `JavaPairRDD` 上没有 `saveAsSequenceFile()` 方法。我们要使用 Spark 保存自定义 Hadoop 格式的功能来实现。5.2.6 节会展示如何使用 Java 以 SequenceFile 保存数据。

5.2.5　对象文件

对象文件看起来就像是对 SequenceFile 的简单封装，它允许存储只包含值的 RDD。和 SequenceFile 不一样的是，对象文件是使用 Java 序列化写出的。

 如果你修改了你的类——比如增减了几个字段——已经生成的对象文件就不再可读了。对象文件使用 Java 序列化，它对兼容同一个类的不同版本有一

定程度的支持，但是需要程序员去实现。

对对象文件使用 Java 序列化有几个要注意的地方。首先，和普通的 SequenceFile 不同，对于同样的对象，对象文件的输出和 Hadoop 的输出不一样。其次，与其他文件格式不同的是，对象文件通常用于 Spark 作业间的通信。最后，Java 序列化有可能相当慢。

要保存对象文件，只需在 RDD 上调用 `saveAsObjectFile` 就行了。读回对象文件也相当简单：用 SparkContext 中的 `objectFile()` 函数接收一个路径，返回对应的 RDD。

了解了关于使用对象文件的这些注意事项，你可能想知道为什么会有人要用它。使用对象文件的主要原因是它们可以用来保存几乎任意对象而不需要额外的工作。

对象文件在 Python 中无法使用，不过 Python 中的 RDD 和 SparkContext 支持 `saveAsPickleFile()` 和 `pickleFile()` 方法作为替代。这使用了 Python 的 pickle 序列化库。不过，对象文件的注意事项同样适用于 pickle 文件：pickle 库可能很慢，并且在修改类定义后，已经生产的数据文件可能无法再读出来。

5.2.6 Hadoop输入输出格式

除了 Spark 封装的格式之外，也可以与任何 Hadoop 支持的格式交互。Spark 支持新旧两套 Hadoop 文件 API，提供了很大的灵活性。[4]

> 4Hadoop 在演进过程中增加了一套新的 MapReduce API，不过有些库仍然使用旧的那套。

1. 读取其他Hadoop输入格式

要使用新版的 Hadoop API 读入一个文件，需要告诉 Spark 一些东西。`newAPIHadoopFile` 接收一个路径以及三个类。第一个类是“格式”类，代表输入格式。相似的函数 `hadoopFile()` 则用于使用旧的 API 实现的 Hadoop 输入格式。第二个类是键的类，最后一个类是值的类。如果需要设定额外的 Hadoop 配置属性，也可以传入一个 `conf` 对象。

`KeyValueTextInputFormat` 是最简单的 Hadoop 输入格式之一，可以用于从文本文件中读取键值对数据（如例 5-24 所示）。每一行都会被独立处理，键和值之间用制表符隔开。这个格式存在于

Hadoop 中，所以无需向工程中添加额外的依赖就能使用它。

例 5-24：在 Scala 中使用老式 API 读取 KeyValueTextInputFormat()

```
val input = sc.hadoopFile[Text, Text, KeyValueTextInputFormat](inputFile).map{
  case (x, y) => (x.toString, y.toString)
}
```

我们学习了通过读取文本文件并加以解析以读取 JSON 数据的方法。事实上，我们也可以使用自定义 Hadoop 输入格式来读取 JSON 数据。该示例需要设置一些额外的压缩选项，我们暂且跳过关于设置压缩选项的细节。Twitter 的 Elephant Bird 包（https://github.com/twitter/elephant-bird）支持很多种数据格式，包括 JSON、Lucene、Protocol Buffer 相关的格式等。这个包也适用于新旧两种 Hadoop 文件 API。为了展示如何在 Spark 中使用新式 Hadoop API，我们来看一个使用 `Lzo JsonInputFormat` 读取 LZO 算法压缩的 JSON 数据的例子。

例 5-25：在 Scala 中使用 Elephant Bird 读取 LZO 算法压缩的 JSON 文件

```
val input = sc.newAPIHadoopFile(inputFile, classOf[LzoJsonInputFormat],
  classOf[LongWritable], classOf[MapWritable], conf)
// "输入"中的每个MapWritable代表一个JSON对象
```


LZO 的支持要求你先安装 `hadoop-lzo` 包，并放到 Spark 的本地库中。如果你使用 Debian 包安装，在调用 `spark-submit` 时加上 `--driver-library-path / usr/lib/hadoop/lib/native/ --driver-class-path /usr/lib/hadoop/lib/` 就可以了。

使用旧的 Hadoop API 读取文件在用法上几乎一样，除了需要提供旧式 `InputFormat` 类。Spark 许多自带的封装好的函数（比如 `sequenceFile()`）都是使用旧式 Hadoop API 实现的。

2. 保存Hadoop输出格式

我们对 SequenceFile 已经有了一定的了解，但是在 Java API 中没有

易用的保存 pair RDD 的函数。我们就把这种情况作为展示如何使用旧式 Hadoop 格式的 API 的例子（见例 5-26）；新接口（`saveAsNewAPIHadoopFile`）的调用方法也是类似的。

例 **5-26**：在 Java 保存 SequenceFile

```
public static class ConvertToWritableTypes implements
  PairFunction<Tuple2<String, Integer>, Text, IntWritable> {
  public Tuple2<Text, IntWritable> call(Tuple2<String, Integer> record) {
    return new Tuple2(new Text(record._1), new IntWritable(record._2));
  }
}

JavaPairRDD<String, Integer> rdd = sc.parallelizePairs(input);
JavaPairRDD<Text, IntWritable> result = rdd.mapToPair(new ConvertToWritabl
  SequenceFileOutputFormat.class);
```

3. 非文件系统数据源

除了 `hadoopFile()` 和 `saveAsHadoopFile()` 这一大类函数，还可以使用 `hadoopDataset/saveAsHadoopDataSet` 和 `newAPIHadoopDataset/saveAsNewAPIHadoopDataset` 来访问 Hadoop 所支持的非文件系统的存储格式。例如，许多像 HBase 和 MongoDB 这样的键值对存储都提供了用来直接读取 Hadoop 输入格式的接口。我们可以在 Spark 中很方便地使用这些格式。

`hadoopDataset()` 这一组函数只接收一个 `Configuration` 对象，这个对象用来设置访问数据源所必需的 Hadoop 属性。你要使用与配置 Hadoop MapReduce 作业相同的方式来配置这个对象。所以你应当按照在 MapReduce 中访问这些数据源的使用说明来配置，并把配置对象传给 Spark。比如，5.5.3 节展示了如何使用 `newAPIHadoopDataset` 来从 HBase 中读取数据。

4. 示例：**protocol buffer**

Protocol buffer（简称 PB，https://github.com/google/protobuf）[5] 最早由 Google 开发，用于内部的远程过程调用（RPC），已经开源。PB 是结构化数据，它要求字段和类型都要明确定义。它们是经过优化的，编解码速度快，而且占用空间也很小。比起 XML，PB 能在同样的空间内存储大约 3 到 10 倍的数据，同时编解码速度大约为 XML 的 20 至 100 倍。PB 采用一致化编码，因此有很多种创建一个包含多个 PB 消息的文件的方式。

5有时称为 pb 或 protobuf。

PB 使用领域专用语言来定义，PB 编译器可以生成各种语言的访问函数（包括 Spark 支持的那些语言）。由于 PB 需要占用尽量少的空间，所以它不是“自描述”的，因为对数据描述的编码需要占用额外的空间。这表示当我们需要解析 PB 格式的数据时，需要获取并理解 PB 的定义。

PB 由可选字段、必需字段、重复字段三种字段组成。在解析时，可选字段的缺失不会导致解析失败，而必需字段的缺失则会导致数据解析失败。因此，在往 PB 定义中添加新字段时，最好将新字段设为可选字段，毕竟不是所有人都会同时更新到新版本（即使他们会这样做，你还是有可能需要读取以前的旧数据）。

PB 字段支持许多预定义类型，或者另一个 PB 消息。这些类型包括 `string`、`int32`、`enum` 等。这里将不提供 PB 的完整介绍，如果你感兴趣的话，可以访问 Protocol Buffer 的网站（https://developers.google.com/protocol-buffers）了解更多细节。

例 5-27 研究的是如何从一个简单的 PB 格式中读取许多 `VenueResponse` 对象。`VenueResponse` 是只包含一个重复字段的简单格式，这个字段包含一条带有必需字段、可选字段以及枚举类型字段的 PB 消息。

例 **5-27**：PB 定义示例

```
message Venue {
  required int32 id = 1;
  required string name = 2;
  required VenueType type = 3;
  optional string address = 4;

  enum VenueType {
    COFFEESHOP = 0;
    WORKPLACE = 1;
    CLUB = 2;
    OMNOMNOM = 3;
    OTHER = 4;
  }
}

message VenueResponse {
  repeated Venue results = 1;
}
```

前一节中使用过 Twitter 的 Elephant Bird 库来读取 JSON 数据，它也支持从 PB 中读取和保存数据。下面来看一个写出 `Venues` 的示例，如例 5-28 所示。

例 **5-28**：在 Scala 中使用 Elephant Bird 写出 protocol buffer

```
val job = new Job()
val conf = job.getConfiguration
LzoProtobufBlockOutputFormat.setClassConf(classOf[Places.Venue], conf);
val dnaLounge = Places.Venue.newBuilder()
dnaLounge.setId(1);
dnaLounge.setName("DNA Lounge")
dnaLounge.setType(Places.Venue.VenueType.CLUB)
val data = sc.parallelize(List(dnaLounge.build()))
val outputData = data.map{ pb =>
  val protoWritable = ProtobufWritable.newInstance(classOf[Places.Venue]);
  protoWritable.set(pb)
  (null, protoWritable)
}
outputData.saveAsNewAPIHadoopFile(outputFile, classOf[Text],
  classOf[ProtobufWritable[Places.Venue]],
  classOf[LzoProtobufBlockOutputFormat[ProtobufWritable[Places.Venue]]], c
```

这个示例的完整版本可以在本书的源代码中找到。

 构建工程时，请确保使用的 PB 库的版本与 Spark 相同。在写作本书之际，Spark 使用的版本是 2.5。

5.2.7 文件压缩

在大数据工作中，我们经常需要对数据进行压缩以节省存储空间和网络传输开销。对于大多数 Hadoop 输出格式来说，我们可以指定一种压缩编解码器来压缩数据。我们已经提过，Spark 原生的输入方式（`textFile` 和 `sequenceFile`）可以自动处理一些类型的压缩。在读取压缩后的数据时，一些压缩编解码器可以推测压缩类型。

这些压缩选项只适用于支持压缩的 Hadoop 格式，也就是那些写出到文件系统的格式。写入数据库的 Hadoop 格式一般没有实现压缩支持。如果数据库中有压缩过的记录，那应该是数据库自己配置的。

选择一个输出压缩编解码器可能会对这些数据以后的用户产生巨大影响。对于像 Spark 这样的分布式系统，我们通常会尝试从多个不同机

器上一起读入数据。要实现这种情况，每个工作节点都必须能够找到一条新记录的开端。有些压缩格式会使这变得不可能，而必须要单个节点来读入所有数据，这就很容易产生性能瓶颈。可以很容易地从多个节点上并行读取的格式被称为“可分割”的格式。表 5-3 列出了可用的压缩选项。

表5-3：压缩选项

[illegible]	[illegible]	Hadoop 压缩编解码器	[illegible]
gzip	[illegible]	org.apache.hadoop.io.compress.GzipCodec	
[illegible]	[illegible]	[illegible]lzo.LzoCodec	需要在每个节点上安装LZO
[illegible]	[illegible]	[illegible]compress.BZip2Codec	为可分割版本使用纯Java
[illegible]	[illegible]	[illegible]compress.DefaultCodec	Hadoop 的默认压缩编解码器
Snappy	[illegible]	[illegible]	Snappy 有纯 Java 的移植版，但是在 Spark/Hadoop 中不能用

6取决于所使用的库。

尽管 Spark 的 `textFile()` 方法可以处理压缩过的输入，但即使输入数据被以可分割读取的方式压缩，Spark 也不会打开 `splittable`。因此，如果你要读取单个压缩过的输入，最好不要考虑使用 Spark 的封装，而是使用 `newAPIHadoopFile` 或者 `hadoopFile`，并指定正确的压缩编解码器。

有些输入格式（例如 SequenceFile）允许我们只压缩键值对数据中的值，这在查询时很有用。其他一些输入格式也有自己的压缩控制：比如，Twitter 的 Elephant Bird 包中的许多格式都可以使用 LZO 算法压缩的数据。

5.3　文件系统

Spark 支持读写很多种文件系统，可以使用任何我们想要的文件格式。

5.3.1　本地/“常规”文件系统

Spark 支持从本地文件系统中读取文件，不过它要求文件在集群中所有节点的相同路径下都可以找到。

一些像 NFS、AFS 以及 MapR 的 NFS layer 这样的网络文件系统会把

文件以常规文件系统的形式暴露给用户。如果你的数据已经在这些系统中，那么你只需要指定输入为一个 file:// 路径；只要这个文件系统挂载在每个节点的同一个路径下，Spark 就会自动处理（如例 5-29 所示）。

例 **5-29**：在 Scala 中从本地文件系统读取一个压缩的文本文件

```
val rdd = sc.textFile("file:///home/holden/happypandas.gz")
```

如果文件还没有放在集群中的所有节点上，你可以在驱动器程序中从本地读取该文件而无需使用整个集群，然后再调用 `parallelize` 将内容分发给工作节点。不过这种方式可能会比较慢，所以推荐的方法是将文件先放到像 HDFS、NFS、S3 等共享文件系统上。

5.3.2　Amazon S3

用 Amazon S3 来存储大量数据正日益流行。当计算节点部署在 Amazon EC2 上的时候，使用 S3 作为存储尤其快，但是在需要通过公网访问数据时性能会差很多。

要在 Spark 中访问 S3 数据，你应该首先把你的 S3 访问凭据设置为 `AWS_ACCESS_KEY_ID` 和 `AWS_SECRET_ACCESS_KEY` 环境变量。你可以从 `Amazon Web Service` 控制台创建这些凭据。接下来，将一个以 `s3n://` 开头的路径以 `s3n://bucket/path-within-bucket` 的形式传给 Spark 的输入方法。和其他所有文件系统一样，Spark 也能在 S3 路径中支持通配字符，例如 `s3n://bucket/my-Files/*.txt`。

如果你从 Amazon 那里得到 S3 访问权限错误，请确保你指定了访问密钥的账号对数据桶有“read”（读）和“list”（列表）的权限。Spark 需要列出桶内的内容，来找到想要读取的数据。

5.3.3　HDFS

Hadoop 分布式文件系统（HDFS）是一种广泛使用的文件系统，Spark 能够很好地使用它。HDFS 被设计为可以在廉价的硬件上工作，有弹性地应对节点失败，同时提供高吞吐量。Spark 和 HDFS 可以部署在同一批机器上，这样 Spark 可以利用数据分布来尽量避免一些网络开销。

在 Spark 中使用 HDFS 只需要将输入输出路径指定为 hdfs://master:port/path 就够了。



HDFS 协议随 Hadoop 版本改变而变化，因此如果你使用的 Spark 是依赖于另一个版本的 Hadoop 编译的，那么读取会失败。默认情况下，Spark 基于 Hadoop 1.0.4 编译 7。如果从源代码编译，你可以在环境变量中指定 `SPARK_HADOOP_VERSION=` 来基于另一个版本的 Hadoop 进行编译；也可以直接下载预编译好的 Spark 版本。你可以根据运行 `hadoop version` 的结果来获得环境变量要设置的值。

7自 Spark 1.4.0 起，Spark 默认的 Hadoop 版本已升级至 2.2.0。——译者注

5.4　Spark SQL中的结构化数据

Spark SQL 是在 Spark 1.0 中新加入 Spark 的组件，并快速成为了 Spark 中较受欢迎的操作结构化和半结构化数据的方式。结构化数据指的是有结构信息的数据——也就是所有的数据记录都具有一致字段结构的集合。Spark SQL 支持多种结构化数据源作为输入，而且由于 Spark SQL 知道数据的结构信息，它还可以从这些数据源中只读出所需字段。第 9 章将更详细地讲解 Spark SQL，现在我们只展示如何使用它从一些常见数据源中读取数据。

在各种情况下，我们把一条 SQL 查询给 Spark SQL，让它对一个数据源执行查询（选出一些字段或者对字段使用一些函数），然后得到由 `Row` 对象组成的 RDD，每个 `Row` 对象表示一条记录。在 Java 和 Scala 中，`Row` 对象的访问是基于下标的。每个 `Row` 都有一个 `get()` 方法，会返回一个一般类型让我们可以进行类型转换。另外还有针对常见基本类型的专用 `get()` 方法（例如 `getFloat()`、`getInt()`、`getLong()`、`getString()`、`getShort()`、`getBoolean()` 等）。在 Python 中，可以使用 `row[column_number]` 以及 `row.column_name` 来访问元素。

5.4.1　Apache Hive

Apache Hive 是 Hadoop 上的一种常见的结构化数据源。Hive 可以在 HDFS 内或者在其他存储系统上存储多种格式的表。这些格式从普通文本到列式存储格式，应有尽有。Spark SQL 可以读取 Hive 支持的

任何表。

要把 Spark SQL 连接到已有的 Hive 上，你需要提供 Hive 的配置文件。你需要将 hive-site.xml 文件复制到 Spark 的 ./conf/ 目录下。这样做好之后，再创建出 `HiveContext` 对象，也就是 Spark SQL 的入口，然后你就可以使用 Hive 查询语言（HQL）来对你的表进行查询，并以由行组成的 RDD 的形式拿到返回数据，如例 5-30 至例 5-32 所示。

例 **5-30**：用 Python 创建 HiveContext 并查询数据

```
from pyspark.sql import HiveContext

hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("SELECT name, age FROM users")
firstRow = rows.first()
print firstRow.name
```

例 **5-31**：用 Scala 创建 HiveContext 并查询数据

```
import org.apache.spark.sql.hive.HiveContext

val hiveCtx = new org.apache.spark.sql.hive.HiveContext(sc)
val rows = hiveCtx.sql("SELECT name, age FROM users")
val firstRow = rows.first()
println(firstRow.getString(0)) // 字段0是name字段
```

例 **5-32**：用 Java 创建 HiveContext 并查询数据

```
import org.apache.spark.sql.hive.HiveContext;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SchemaRDD;

HiveContext hiveCtx = new HiveContext(sc);
SchemaRDD rows = hiveCtx.sql("SELECT name, age FROM users");
Row firstRow = rows.first();
System.out.println(firstRow.getString(0)); // 字段0是name字段
```

我们会在 9.3.1 节更详细地介绍如何从 Hive 中读取数据。

5.4.2　JSON

如果你有记录间结构一致的 JSON 数据，Spark SQL 也可以自动推断出它们的结构信息，并将这些数据读取为记录，这样就可以使得提取字段的操作变得很简单。要读取 JSON 数据，首先需要和使用 Hive 一样创建一个 `HiveContext`。（不过在这种情况下我们不需要安装好 Hive，也就是说你也不需要 hive-site.xml 文件。）然后使用 `HiveContext.jsonFile` 方法来从整个文件中获取由 `Row` 对象组成的 RDD。除了使用整个 `Row` 对象，你也可以将 RDD 注册为一张表，然后从中选出特定的字段。例如，假设有一个包含推文的 JSON 文件，格式如例 5-33 所示，每行一条记录。

例 **5-33**：JSON 中的示例推文

```
{"user": {"name": "Holden", "location": "San Francisco"}, "text": "Nice da
{"user": {"name": "Matei", "location": "Berkeley"}, "text": "Even nicer he
```

我们可以读取这些数据，只从中选取 `name`（用户名）和 `text`（文本）字段，如例 5-34 至例 5-36 所示。

例 **5-34**：在 Python 中使用 Spark SQL 读取 JSON 数据

```
tweets = hiveCtx.jsonFile("tweets.json")
tweets.registerTempTable("tweets")
results = hiveCtx.sql("SELECT user.name, text FROM tweets")
```

例 **5-35**：在 Scala 中使用 Spark SQL 读取 JSON 数据

```
val tweets = hiveCtx.jsonFile("tweets.json")
tweets.registerTempTable("tweets")
val results = hiveCtx.sql("SELECT user.name, text FROM tweets")
```

例 **5-36**：在 Java 中使用 Spark SQL 读取 JSON 数据

```
SchemaRDD tweets = hiveCtx.jsonFile(jsonFile);
tweets.registerTempTable("tweets");
SchemaRDD results = hiveCtx.sql("SELECT user.name, text FROM tweets");
```

我们会在 9.3.3 节对如何使用 Spark SQL 读取 JSON 数据并访问其结构信息进行深入探讨。此外，Spark SQL 的支持远不限于读取数据，还包括查询数据、以比 RDD 所支持的方式更复杂的方式组合数据、对数据运行自定义函数，这些都将在第 9 章中讲到。

5.5　数据库

通过数据库提供的 Hadoop 连接器或者自定义的 Spark 连接器，Spark 可以访问一些常用的数据库系统。本节来展示四种常见的连接器。

5.5.1　Java数据库连接

Spark 可以从任何支持 Java 数据库连接（JDBC）的关系型数据库中读取数据，包括 MySQL、Postgres 等系统。要访问这些数据，需要构建一个 `org.apache.spark.rdd.JdbcRDD`，将 SparkContext 和其他参数一起传给它。例 5-37 就演示了如何使用 `JdbcRDD` 连接 MySQL 数据库。

例 **5-37**：Scala 中的 JdbcRDD

```
def createConnection() = {
  Class.forName("com.mysql.jdbc.Driver").newInstance();
  DriverManager.getConnection("jdbc:mysql://localhost/test?user=holden");
}

def extractValues(r: ResultSet) = {
  (r.getInt(1), r.getString(2))
}

val data = new JdbcRDD(sc,
  createConnection, "SELECT * FROM panda WHERE ? <= id AND id <= ?",
  lowerBound = 1, upperBound = 3, numPartitions = 2, mapRow = extractValue
println(data.collect().toList)
```

`JdbcRDD` 接收这样几个参数。

- 首先，要提供一个用于对数据库创建连接的函数。这个函数让每个节点在连接必要的配置后创建自己读取数据的连接。
- 接下来，要提供一个可以读取一定范围内数据的查询，以及查询参数中 `lowerBound` 和 `upperBound` 的值。这些参数可

以让 Spark 在不同机器上查询不同范围的数据，这样就不会因尝试在一个节点上读取所有数据而遭遇性能瓶颈。8

- 这个函数的最后一个参数是一个可以将输出结果从 java.sql.ResultSet（http://docs.oracle.com/javase/7/docs/api/java/sql/ResultSet.html）转为对操作数据有用的格式的函数。在例 5-37 中，我们会得到 (Int, String) 对。如果这个参数空缺，Spark 会自动将每行结果转为一个对象数组。

8如果你不知道到底有多少条记录，可以先手动执行一条计数查询，然后根据结果来决定 upperBound 和 lowerBound 的值。

和其他的数据源一样，在使用 JdbcRDD 时，需要确保你的数据库可以应付 Spark 并行读取的负载。如果你想要离线查询数据而不使用在线数据库，可以使用数据库的导出功能，将数据导出为文本文件。

5.5.2 Cassandra

随着 DataStax 开源其用于 Spark 的 Cassandra 连接器（https://github.com/datastax/spark-cassandra-connector），Spark 对 Cassandra 的支持大大提升。这个连接器目前还不是 Spark 的一部分，因此你需要添加一些额外的依赖到你的构建文件中才能使用它。Cassandra 还没有使用 Spark SQL，不过它会返回由 CassandraRow 对象组成的 RDD，这些对象有一部分方法与 Spark SQL 的 Row 对象的方法相同，如例 5-38 和例 5-39 所示。Spark 的 Cassandra 连接器目前只能在 Java 和 Scala 中使用。

例 **5-38**：Cassandra 连接器的 sbt 依赖

```
"com.datastax.spark" %% "spark-cassandra-connector" % "1.0.0-rc5",
"com.datastax.spark" %% "spark-cassandra-connector-java" % "1.0.0-rc5"
```

例 **5-39**：Cassandra 连接器的 Maven 依赖

```
<dependency> <!-- Cassandra -->
  <groupId>com.datastax.spark</groupId>
  <artifactId>spark-cassandra-connector</artifactId>
  <version>1.0.0-rc5</version>
</dependency>
<dependency> <!-- Cassandra -->
  <groupId>com.datastax.spark</groupId>
```

```
  <artifactId>spark-cassandra-connector-java</artifactId>
  <version>1.0.0-rc5</version>
</dependency>
```

跟 Elasticsearch 很像，Cassandra 连接器要读取一个作业属性来决定连接到哪个集群。我们把 spark.cassandra.connection.host 设置为指向 Cassandra 集群。如果有用户名和密码的话，则需要分别设置 spark.cassandra.auth.username 和 spark.cassandra.auth.password。假定你只有一个 Cassandra 集群要连接，可以在创建 SparkContext 时就把这些都设好，如例 5-40 和例 5-41 所示。

例 **5-40**：在 Scala 中配置 Cassandra 属性

```
val conf = new SparkConf(true)
        .set("spark.cassandra.connection.host", "hostname")

val sc = new SparkContext(conf)
```

例 **5-41**：在 Java 中配置 Cassandra 属性

```
SparkConf conf = new SparkConf(true)
  .set("spark.cassandra.connection.host", cassandraHost);
JavaSparkContext sc = new JavaSparkContext(
  sparkMaster, "basicquerycassandra", conf);
```

Datastax 的 Cassandra 连接器使用 Scala 中的隐式转换来为 SparkContext 和 RDD 提供一些附加函数。让我们引入这些隐式转换，并尝试读取一些数据（如例 5-42 所示）。

例 **5-42**：在 Scala 中将整张键值对表读取为 RDD

```
// 为SparkContext和RDD提供附加函数的隐式转换
import com.datastax.spark.connector._

// 将整张表读为一个RDD。假设你的表test的创建语句为
// CREATE TABLE test.kv(key text PRIMARY KEY, value int);
val data = sc.cassandraTable("test" , "kv")
// 打印出value字段的一些基本统计。
data.map(row => row.getInt("value")).stats()
```

在 Java 中，由于没有隐式转换，所以需要显式地转换 SparkContext 对象和 RDD 来实现这样的功能（如例 5-43 所示）。

例 **5-43**：在 Java 中将整张键值对表读取为 RDD

```
import com.datastax.spark.connector.CassandraRow;
import static com.datastax.spark.connector.CassandraJavaUtil.javaFunctions

// 将整张表读为一个RDD。假设你的表test的创建语句为
// CREATE TABLE test.kv(key text PRIMARY KEY, value int);
JavaRDD<CassandraRow> data = javaFunctions(sc).cassandraTable("test" , "kv
// 打印一些基本统计。
System.out.println(data.mapToDouble(new DoubleFunction<CassandraRow>() {
  public double call(CassandraRow row) { return row.getInt("value"); }
}).stats());
```

除了读取整张表，也可以查询数据集的子集。通过在 `cassandraTable()` 的调用中加上 `where` 子句，可以限制查询的数据，例如 `sc.cassandraTable(...).where("key=?", "panda")`。

Cassandra 连接器支持把多种类型的RDD保存到 Cassandra 中。我们可以直接保存由 `CassandraRow` 对象组成的 RDD，这对于在表之间复制数据比较有用。通过指定列的映射关系，我们也可以存储不是行的形式而是元组和列表的形式的 RDD，如例 5-44 所示。

例 **5-44**：在 Scala 中保存数据到 Cassandra

```
val rdd = sc.parallelize(List(Seq("moremagic", 1)))
rdd.saveToCassandra("test" , "kv", SomeColumns("key", "value"))
```

本节只是简短地介绍了 Cassandra 连接器。要了解更多信息，请查阅该连接器的 GitHub 页面（https://github.com/datastax/spark-cassandra-connector）。

5.5.3 HBase

由于 `org.apache.hadoop.hbase.mapreduce.TableInputFormat` 类的实现，Spark 可以通过 Hadoop 输入格式访问 HBase。这个输入格式会返回键值对数据，其中键的类型为 `org.apache.hadoop.hbase.io.ImmutableBytesWritable`，

而值的类型为 org.apache.hadoop.hbase.client.Result。Result 类包含多种根据列获取值的方法，在其 API 文档（https://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Result.html）中有所描述。

要将 Spark 用于 HBase，你需要使用正确的输入格式调用 SparkContext.newAPIHadoopRDD。

Scala 中的示例如例 5-45 所示。

例 5-45：从 HBase 读取数据的 Scala 示例

```
import org.apache.hadoop.hbase.HBaseConfiguration
import org.apache.hadoop.hbase.client.Result
import org.apache.hadoop.hbase.io.ImmutableBytesWritable
import org.apache.hadoop.hbase.mapreduce.TableInputFormat

val conf = HBaseConfiguration.create()
conf.set(TableInputFormat.INPUT_TABLE, "tablename") // 扫描哪张表

val rdd = sc.newAPIHadoopRDD(
  conf, classOf[TableInputFormat], classOf[ImmutableBytesWritable],classOf
```

TableInputFormat 包含多个可以用来优化对 HBase 的读取的设置项，比如将扫描限制到一部分列中，以及限制扫描的时间范围。你可以在 TableInputFormat 的 API 文档（http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/mapreduce/TableInputFormat.html）中找到这些选项，并在 HBaseConfiguration 中设置它们，然后再把它传给 Spark。

5.5.4 Elasticsearch

Spark 可以使用 Elasticsearch-Hadoop（https://github.com/elastic/elasticsearch-hadoop）从 Elasticsearch 中读写数据。Elasticsearch 是一个开源的、基于 Lucene 的搜索系统。

Elasticsearch 连接器和我们研究过的其他连接器不大一样，它会忽略我们提供的路径信息，而依赖于在 SparkContext 中设置的配置项。Elasticsearch 的 OutputFormat 连接器也没有用到 Spark 所封装的类型，所以我们使用 saveAsHadoopDataSet 来代替，这意味着我们需要手动设置更多属性。让我们通过例 5-46 和例 5-47 来看看如何对 Elasticsearch 读写一些简单的数据。



最新版的 Elasticsearch Spark 连接器用起来更简单，支持返回 Spark SQL 中的行对象。这个连接器仍然是隐藏的，因为行转换还不支持 Elasticsearch 中所有的原生类型。

例 5-46：在 Scala 中使用 Elasticsearch 输出

```
val jobConf = new JobConf(sc.hadoopConfiguration)
jobConf.set("mapred.output.format.class", "org.elasticsearch.hadoop.
mr.EsOutputFormat")
jobConf.setOutputCommitter(classOf[FileOutputCommitter])
jobConf.set(ConfigurationOptions.ES_RESOURCE_WRITE, "twitter/tweets")
jobConf.set(ConfigurationOptions.ES_NODES, "localhost")
FileOutputFormat.setOutputPath(jobConf, new Path("-"))
output.saveAsHadoopDataset(jobConf)
```

例 5-47：在 Scala 中使用 Elasticsearch 输入

```
def mapWritableToInput(in: MapWritable): Map[String, String] = {
  in.map{case (k, v) => (k.toString, v.toString)}.toMap
}

val jobConf = new JobConf(sc.hadoopConfiguration)
jobConf.set(ConfigurationOptions.ES_RESOURCE_READ, args(1))
jobConf.set(ConfigurationOptions.ES_NODES, args(2))
val currentTweets = sc.hadoopRDD(jobConf,
  classOf[EsInputFormat[Object, MapWritable]], classOf[Object],
  classOf[MapWritable])
// 仅提取map
// 将MapWritable[Text, Text]转为Map[String, String]
val tweets = currentTweets.map{ case (key, value) => mapWritableToInput(va
```

和其他连接器相比，Elasticsearch 连接器有点复杂，但这也是对于如何操作这类连接器的一个有效参考。



就输出而言，Elasticsearch 可以进行映射推断，但是偶尔会推断出不正确的数据类型，因此如果你要存储字符串以外的数据类型，最好明确指定类型映射（https://www.elastic.co/guide/en/elasticsearch/reference/current/indices-put-mapping.html）。

5.6　总结

在本章结束之际，你应该已经能够将数据读取到 Spark 中，并将计算结果以你所希望的方式存储起来。我们调查了数据可以使用的一些不同格式，一些压缩选项以及它们对应的数据处理的方式。现在我们已经掌握了读取和保存大规模数据集的方法，后续章节会介绍一些用来编写更高效更强大的 Spark 程序的方法。

第 6 章　Spark 编程进阶

6.1　简介

本章介绍前几章没有提及的 Spark 编程的各种进阶特性，会介绍两种类型的共享变量：累加器（accumulator）与广播变量（broadcast variable）。累加器用来对信息进行聚合，而广播变量用来高效分发较大的对象。在已有的 RDD 转化操作的基础上，我们为类似查询数据库这样需要很大配置代价的任务引入了批操作。为了扩展可用的工具范围，本章会介绍 Spark 与外部程序交互的方式，比如如何与用 R 语言编写的脚本进行交互。

本章会使用业余无线电操作者的呼叫日志作为输入，构建出一个完整的示例应用。这些日志至少包含联系过的站点的呼号。呼号是由国家分配的，每个国家都有自己的呼号号段，所以我们可以根据呼号查到对应的国家。有一些呼叫日志也包含操作者的地理位置，用来帮助确定距离。例 6-1 展示了一段示例日志。本书的示例代码仓库中包含一个需要从呼叫日志中查询并进行处理的呼号列表。

例 **6-1**：一条 JSON 格式的呼叫日志示例，其中某些字段已省略

```
{"address":"address here", "band":"40m","callsign":"KK6JLK","city":"SUNNYV
"contactlat":"37.384733","contactlong":"-122.032164",
"county":"Santa Clara","dxcc":"291","fullname":"MATTHEW McPherrin",
"id":57779,"mode":"FM","mylat":"37.751952821","mylong":"-122.4208688735",.
```

要用到的第一个 Spark 特性就是共享变量。共享变量是一种可以在 Spark 任务中使用的特殊类型的变量。在示例中，我们使用 Spark 共享变量来对非严重错误的情况进行计数，以及分发一张巨大的查询表。

当任务需要很长时间进行配置，譬如需要创建数据库连接或者随机数生成器时，在多个数据元素间共享一次配置就会比较有效率。由于需要用到远程呼号查询数据库，所以会讨论如何基于分区进行操作以重用数据库连接的配置工作。

除了 Spark 直接支持的语言外，系统还可以调用用别的语言写出来的

程序。本章会介绍如何使用与 Spark 语言无关的方法 `pipe()` 来与其他程序通过标准输入和标准输出进行交互。我们会使用 `pipe()` 方法来访问 R 语言的库，以计算业余电台操作者每次联系的距离。

最后，和操作键值对类似，Spark 也有专门用来操作数值数据的方法。下面通过用业余电台呼叫日志计算出来的距离移除异常值的示例，展示如何使用这些方法。

6.2　累加器

通常在向 Spark 传递函数时，比如使用 `map()` 函数或者用 `filter()` 传条件时，可以使用驱动器程序中定义的变量，但是集群中运行的每个任务都会得到这些变量的一份新的副本，更新这些副本的值也不会影响驱动器中的对应变量。Spark 的两个共享变量，累加器与广播变量，分别为结果聚合与广播这两种常见的通信模式突破了这一限制。

第一种共享变量，即累加器，提供了将工作节点中的值聚合到驱动器程序中的简单语法。累加器的一个常见用途是在调试时对作业执行过程中的事件进行计数。例如，假设我们在从文件中读取呼号列表对应的日志，同时也想知道输入文件中有多少空行（也许不希望在有效输入中看到很多这样的行）。例 6-2 至例 6-4 展示了这一场景。

例 **6-2**：在 Python 中累加空行

```
file = sc.textFile(inputFile)
# 创建Accumulator[Int]并初始化为0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines # 访问全局变量
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
callSigns.saveAsTextFile(outputDir + "/callsigns")
print "Blank lines: %d" % blankLines.value
```

例 **6-3**：在 Scala 中累加空行

```
val sc = new SparkContext(...)
val file = sc.textFile("file.txt")
```

```
val blankLines = sc.accumulator(0) // 创建Accumulator[Int]并初始化为0

val callSigns = file.flatMap(line => {
  if (line == "") {
    blankLines += 1 // 累加器加1
  }
  line.split(" ")
})

callSigns.saveAsTextFile("output.txt")
println("Blank lines: " + blankLines.value)
```

例 **6-4**：在 Java 中累加空行

```
JavaRDD<String> rdd = sc.textFile(args[1]);

final Accumulator<Integer> blankLines = sc.accumulator(0);
JavaRDD<String> callSigns = rdd.flatMap(
  new FlatMapFunction<String, String>() { public Iterable<String> call(Str
      if (line.equals("")) {
        blankLines.add(1);
      }
      return Arrays.asList(line.split(" "));
  }});

callSigns.saveAsTextFile("output.txt")
System.out.println("Blank lines: "+ blankLines.value());
```

在这些示例中，我们创建了一个叫作 `blankLines` 的 `Accumulator[Int]` 对象，然后在输入中看到一个空行时就对其加 1。执行完转化操作之后，就打印出累加器中的值。注意，只有在运行 `saveAsTextFile()` 行动操作后才能看到正确的计数，因为行动操作前的转化操作 `flatMap()` 是惰性的，所以作为计算副产品的累加器只有在惰性的转化操作 `flatMap()` 被 `saveAsTextFile()` 行动操作强制触发时才会开始求值。

当然，也可以使用 `reduce()` 这样的行动操作将整个 RDD 中的值都聚合到驱动器中。只是我们有时希望使用一种更简单的方法来对那些与 RDD 本身的范围和粒度不一样的值进行聚合。聚合可以发生在 RDD 进行转化操作的过程中。在前面的例子中，我们使用累加器在读取数据时对错误进行计数，而没有分别使用 `filter()` 和 `reduce()`。

总结起来，累加器的用法如下所示。

- 通过在驱动器中调用 `SparkContext.accumulator(initialValue)` 方法，创建出存有初始值的累加器。返回值为 `org.apache.spark.Accumulator[T]` 对象，其中 T 是初始值 `initialValue` 的类型。

- Spark 闭包里的执行器代码可以使用累加器的 `+=` 方法（在 Java 中是 `add`）增加累加器的值。

- 驱动器程序可以调用累加器的 `value` 属性（在 Java 中使用 `value()` 或 `setValue()`）来访问累加器的值。

注意，工作节点上的任务不能访问累加器的值。从这些任务的角度来看，累加器是一个只写变量。在这种模式下，累加器的实现可以更加高效，不需要对每次更新操作进行复杂的通信。

这里展示的计数在很多时候都非常方便，比如有多个值需要跟踪时，或者当某个值需要在并行程序的多个地方增长时（比如你可能需要对程序中调用 JSON 解析库的次数进行计数）。例如，我们一般预期数据的一小部分是毁坏的，或者允许后端有一定的失败数次。为了防止产生含有过多错误的垃圾输出，可以使用累加器对有效记录和无效记录分别进行计数。累加器的值只有在驱动器程序中可以访问，所以检查也应当在驱动器程序中完成。

继续之前的示例，现在可以验证呼号，并且只有在大部分输入有效时才输出。国际电信联盟在 19 号文件中对业余电台的呼号格式进行了规范，我们可以根据这一规范，使用正则表达式来验证呼号，如例 6-5 所示。

例 **6-5**：在 Python 使用累加器进行错误计数

```
# 创建用来验证呼号的累加器
validSignCount = sc.accumulator(0)
invalidSignCount = sc.accumulator(0)

def validateSign(sign):
    global validSignCount, invalidSignCount
    if re.match(r"\A\d?[a-zA-Z]{1,2}\d{1,4}[a-zA-Z]{1,3}\Z", sign):
        validSignCount += 1
        return True
    else:
        invalidSignCount += 1
        return False
```

```
# 对与每个呼号的联系次数进行计数
validSigns = callSigns.filter(validateSign)
contactCount = validSigns.map(lambda sign: (sign, 1)).reduceByKey(lambda

# 强制求值计算计数
contactCount.count()
if invalidSignCount.value < 0.1 * validSignCount.value:
    contactCount.saveAsTextFile(outputDir + "/contactCount")
else:
    print "Too many errors: %d in %d" % (invalidSignCount.value, validSign
value)
```

6.2.1　累加器与容错性

Spark 会自动重新执行失败的或较慢的任务来应对有错误的或者比较慢的机器。例如，如果对某分区执行 `map()` 操作的节点失败了，Spark 会在另一个节点上重新运行该任务。即使该节点没有崩溃，而只是处理速度比别的节点慢很多，Spark 也可以抢占式地在另一个节点上启动一个“投机”（speculative）型的任务副本，如果该任务更早结束就可以直接获取结果。即使没有节点失败，Spark 有时也需要重新运行任务来获取缓存中被移除出内存的数据。因此最终结果就是同一个函数可能对同一个数据运行了多次，这取决于集群发生了什么。

这种情况下累加器要怎么处理呢？实际结果是，对于要在行动操作中使用的累加器，**Spark**只会把每个任务对各累加器的修改应用一次。因此，如果想要一个无论在失败还是重复计算时都绝对可靠的累加器，我们必须把它放在 `foreach()` 这样的行动操作中。

对于在 **RDD** 转化操作中使用的累加器，就不能保证有这种情况了。转化操作中累加器可能会发生不止一次更新。举个例子，当一个被缓存下来但是没有经常使用的 RDD 在第一次从 LRU 缓存中被移除并又被重新用到时，这种非预期的多次更新就会发生。这会强制 RDD 根据其谱系进行重算，而副作用就是这也会使得谱系中的转化操作里的累加器进行更新，并再次发送到驱动器中。在转化操作中，累加器通常只用于调试目的。

尽管将来版本的 Spark 可能会把这一行为改成只更新一次累加器的结果，但当前版本（1.2.0）确实会进行多次更新，因此转化操作中的累加器最好只在调试时使用。

6.2.2　自定义累加器

到目前为止，我们学习了如何使用加法操作 Spark 的一种累加器类型整型（`Accumulator[Int]`）。Spark 还直接支持 `Double`、`Long` 和 `Float` 型的累加器。除此以外，Spark 也引入了自定义累加器和聚合操作的 API（比如找到要累加的值中的最大值，而不是把这些值加起来）。自定义累加器需要扩展 `AccumulatorParam`，这在 Spark API 文档（http://spark.apache.org/docs/latest/api/scala/index.html#package）中有所介绍。只要该操作同时满足交换律和结合律，就可以使用任意操作来代替数值上的加法。比如除了跟踪总和，还可以跟踪数据的最大值。



如果对于任意的 a 和 b，有 $a\ op\ b = b\ op\ a$，就说明操作 op 满足交换律。如果对于任意的 a、b 和 c，有 $(a\ op\ b)\ op\ c = a\ op\ (b\ op\ c)$，就说明操作 op 满足结合律。例如，`sum` 和 `max` 既满足交换律又满足结合律，是 Spark 累加器中的常用操作。

6.3　广播变量

Spark 的第二种共享变量类型是广播变量，它可以让程序高效地向所有工作节点发送一个较大的只读值，以供一个或多个 Spark 操作使用。比如，如果你的应用需要向所有节点发送一个较大的只读查询表，甚至是机器学习算法中的一个很大的特征向量，广播变量用起来都很顺手。

前面提过，Spark 会自动把闭包中所有引用到的变量发送到工作节点上。虽然这很方便，但也很低效。原因有二：首先，默认的任务发射机制是专门为小任务进行优化的；其次，事实上你可能会在多个并行操作中使用同一个变量，但是 Spark 会为每个操作分别发送。举个例子，假设要写一个 Spark 程序，通过呼号的前缀来查询对应的国家。虽然前缀的长度不一，但由于每个国家都使用各自的业余呼号前缀，所以这种方法还是可行的。如果用 Spark 直接实现，则代码就如例 6-6 所示。

例 **6-6**：在 Python 中查询国家

```
# 查询RDD contactCounts中的呼号的对应位置。将呼号前缀
# 读取为国家代码来进行查询
signPrefixes = loadCallSignTable()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
```

```
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

这个程序可以运行，但是如果表更大（比如表中不是呼号而是 IP 地址），signPrefixes 很容易就会达到数 MB 大小，从主节点为每个任务发送一个这样的数组就会代价巨大。而且，如果之后还要再次使用 signPrefixes 这个对象（可能还要在 file2.txt 上运行同样的代码），则还需要向每个节点再发送一遍。

我们可以把 signPrefixes 变为广播变量来解决这一问题。广播变量其实就是类型为 spark.broadcast.Broadcast[T] 的一个对象，其中存放着类型为 T 的值。可以在任务中通过对 Broadcast 对象调用 value 来获取该对象的值。这个值只会被发送到各节点一次，使用的是一种高效的类似 BitTorrent 的通信机制。

使用广播变量后，先前的例子就改为了如例 6-7 至例 6-9 所示那样。

例 **6-7**：在 Python 中使用广播变量查询国家

```
# 查询RDD contactCounts中的呼号的对应位置。将呼号前缀
# 读取为国家代码来进行查询
signPrefixes = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))

countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

例 **6-8**：在 Scala 中使用广播变量查询国家

```
// 查询RDD contactCounts中的呼号的对应位置。将呼号前缀
// 读取为国家代码来进行查询
val signPrefixes = sc.broadcast(loadCallSignTable())
val countryContactCounts = contactCounts.map{case (sign, count) =>
  val country = lookupInArray(sign, signPrefixes.value)
```

```
  (country, count)
}.reduceByKey((x, y) => x + y)
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

例 **6-9**：在 Java 中使用广播变量查询国家

```
// 读入呼号表
// 查询RDD contactCounts中的呼号对应的国家
final Broadcast<String[]> signPrefixes = sc.broadcast(loadCallSignTable())
JavaPairRDD<String, Integer> countryContactCounts = contactCounts.mapToPai
  new PairFunction<Tuple2<String, Integer>, String, Integer> (){
    public Tuple2<String, Integer> call(Tuple2<String, Integer> callSignCo
      String sign = callSignCount._1();
      String country = lookupCountry(sign, signPrefixes.value());
      return new Tuple2(country, callSignCount._2());
    }}).reduceByKey(new SumInts());
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt");
```

如以上示例所示，使用广播变量的过程很简单。

(1) 通过对一个类型 `T` 的对象调用 `SparkContext.broadcast` 创建出一个 `Broadcast[T]` 对象。任何可序列化的类型都可以这么实现。

(2) 通过 `value` 属性访问该对象的值（在 Java 中为 `value()` 方法）。

(3) 变量只会被发到各个节点一次，应作为只读值处理（修改这个值不会影响到别的节点）。

满足只读要求的最容易的使用方式是广播基本类型的值或者引用不可变对象。在这样的情况下，你没有办法修改广播变量的值，除了在驱动器程序的代码中可以修改。但是，有时传一个可变对象可能更为方便与高效。如果你这样做的话，就需要自己维护只读的条件。就像对 `Array[String]` 类型的呼号前缀表所做的那样，必须确保从节点上运行的代码不会尝试去做诸如 `val theArray = broadcastArray.value; theArray(0) = newValue` 这样的事情。当在工作节点上执行时，这一行将 `newValue` 赋给数组的第一个元素，但是只对该工作节点本地的这个数组的副本有效，而不会改变任何其他工作节点上通过 `broadcastArray.value` 所读取到的内容。

广播的优化

当广播一个比较大的值时，选择既快又好的序列化格式是很重要的，因为如果序列化对象的时间很长或者传送花费的时间太久，这段时间很容易就成为性能瓶颈。尤其是，Spark 的 Scala 和 Java API 中默认使用的序列化库为 Java 序列化库，因此它对于除基本类型的数组以外的任何对象都比较低效。你可以使用 `spark.serializer` 属性选择另一个序列化库来优化序列化过程（第 8 章中会讨论如何使用 Kryo 这种更快的序列化库），也可以为你的数据类型实现自己的序列化方式（对 Java 对象使用 `java.io.Externalizable` 接口实现序列化，或使用 `reduce()` 方法为 Python 的 pickle 库定义自定义的序列化）。

6.4　基于分区进行操作

基于分区对数据进行操作可以让我们避免为每个数据元素进行重复的配置工作。诸如打开数据库连接或创建随机数生成器等操作，都是我们应当尽量避免为每个元素都配置一次的工作。Spark 提供基于分区的 `map` 和 `foreach`，让你的部分代码只对 RDD 的每个分区运行一次，这样可以帮助降低这些操作的代价。

回到呼号的示例程序中来，我们有一个在线的业余电台呼号数据库，可以用这个数据库查询日志中记录过的联系人呼号列表。通过使用基于分区的操作，可以在每个分区内共享一个数据库连接池，来避免建立太多连接，同时还可以重用 JSON 解析器。如例 6-10 至例 6-12 所示，使用 `mapPartitions` 函数获得输入 RDD 的每个分区中的元素迭代器，而需要返回的是执行结果的序列的迭代器。

例 **6-10**：在 Python 中使用共享连接池

```
def processCallSigns(signs):
    """使用连接池查询呼号"""
    # 创建一个连接池
    http = urllib3.PoolManager()
    # 与每条呼号记录相关联的URL
    urls = map(lambda x: "http://73s.com/qsos/%s.json" % x, signs)
    # 创建请求（非阻塞）
    requests = map(lambda x: (x, http.request('GET', x)), urls)
    # 获取结果
    result = map(lambda x: (x[0], json.loads(x[1].data)), requests)
    # 删除空的结果并返回
    return filter(lambda x: x[1] is not None, result)
```

```
def fetchCallSigns(input):
    """获取呼号"""
    return input.mapPartitions(lambda callSigns : processCallSigns(callSig

contactsContactList = fetchCallSigns(validSigns)
```

例 6-11：在 Scala 中使用共享连接池与 JSON 解析器

```
val contactsContactLists = validSigns.distinct().mapPartitions{
  signs =>
  val mapper = createMapper()
  val client = new HttpClient()
  client.start()
  // 创建http请求
  signs.map {sign =>
    createExchangeForSign(sign)
 // 获取响应
  }.map{ case (sign, exchange) =>
      (sign, readExchangeCallLog(mapper, exchange))
  }.filter(x => x._2 != null) // 删除空的呼叫日志
}
```

例 6-12：在 Java 中使用共享连接池与 JSON 解析器

```
// 使用mapPartitions重用配置工作
JavaPairRDD<String, CallLog[]> contactsContactLists =
  validCallSigns.mapPartitionsToPair(
  new PairFlatMapFunction<Iterator<String>, String, CallLog[]>() {
    public Iterable<Tuple2<String, CallLog[]>> call(Iterator<String> input
      // 列出结果
      ArrayList<Tuple2<String, CallLog[]>> callsignLogs = new ArrayList<>
      ArrayList<Tuple2<String, ContentExchange>> requests = new ArrayList<
      ObjectMapper mapper = createMapper();
      HttpClient client = new HttpClient();
      try {
        client.start();
        while (input.hasNext()) {
          requests.add(createRequestForSign(input.next(), client));
        }
        for (Tuple2<String, ContentExchange> signExchange : requests) {
          callsignLogs.add(fetchResultFromRequest(mapper, signExchange));
        }
      } catch (Exception e) {
      }
      return callsignLogs;
    }});
System.out.println(StringUtils.join(contactsContactLists.collect(), ","));
```

当基于分区操作 RDD 时，Spark 会为函数提供该分区中的元素的迭代器。返回值方面，也返回一个迭代器。除 `mapPartitions()` 外，Spark 还有一些别的基于分区的操作符，列在了表 6-1 中。

表**6-1**：按分区执行的操作符

函数名	调用所提供的	返回的	对于RDD[T]的函数签名
mapPartitions()	该分区中元素的迭代器	返回的元素的迭代器	f: (Iterator[T]) → Iterator[U]
mapPartitionsWithIndex()	分区序号，以及每个分区中的元素的迭代器	返回的元素的迭代器	f: (Int, Iterator[T]) → Iterator[U]
foreachPartitions()	元素迭代器	无	f: (Iterator[T]) → Unit

除了避免重复的配置工作，也可以使用 `mapPartitions()` 避免创建对象的开销。有时需要创建一个对象来将不同类型的数据聚合起来。回忆一下第 3 章中，当计算平均值时，一种方法是将数值 RDD 转为二元组 RDD，以在归约过程中追踪所处理的元素个数。现在，可以为每个分区只创建一次二元组，而不用为每个元素都执行这个操作，参见例 6-13 和例 6-14。

例 **6-13**：在 Python 中不使用 `mapPartitions()` 求平均值

```
def combineCtrs(c1, c2):
    return (c1[0] + c2[0], c1[1] + c2[1])

def basicAvg(nums):
    """计算平均值"""
    nums.map(lambda num: (num, 1)).reduce(combineCtrs)
```

例 **6-14**：在 Python 中使用 `mapPartitions()` 求平均值

```
def partitionCtr(nums):
    """计算分区的sumCounter"""
    sumCount = [0, 0]
    for num in nums:
        sumCount[0] += num
        sumCount[1] += 1
    return [sumCount]

def fastAvg(nums):
    """计算平均值"""
    sumCount = nums.mapPartitions(partitionCtr).reduce(combineCtrs)
    return sumCount[0] / float(sumCount[1])
```

6.5　与外部程序间的管道

有三种可用的语言供你选择，这可能已经满足了你用来编写 Spark 应用的几乎所有需求。但是，如果 Scala、Java 以及 Python 都不能实现你需要的功能，那么 Spark 也为这种情况提供了一种通用机制，可以将数据通过管道传给用其他语言编写的程序，比如 R 语言脚本。

Spark 在 RDD 上提供 `pipe()` 方法。Spark 的 `pipe()` 方法可以让我们使用任意一种语言实现 Spark 作业中的部分逻辑，只要它能读写 Unix 标准流就行。通过 `pipe()`，你可以将 RDD 中的各元素从标准输入流中以字符串形式读出，并对这些元素执行任何你需要的操作，然后把结果以字符串的形式写入标准输出——这个过程就是 RDD 的转化操作过程。这种接口和编程模型有较大的局限性，但是有时候这恰恰是你想要的，比如在 `map` 或 `filter` 操作中使用某些语言原生的函数。

有时候，由于你已经写好并测试好了一些很复杂的软件，所以会希望把 RDD 中的内容通过管道交给这些外部程序或者脚本来进行处理并重用。很多数据科学家都用 R1 写好的代码 **2**，可以通过 `pipe()` 与 R 程序进行交互。

> 1SparkR 在 Spark 1.4.0 中已成为 Spark 的一部分。——译者注

> 2SparkR 项目也使用 Spark 提供了一个轻量级前端，以便在 R 中使用 Spark。

在例 6-15 中，我们使用一个 R 语言的库来计算所有联系人的距离。程序会把 RDD 的每个元素都以换行符作为分隔符写出去，而那个 R 程序输出的每一行都是字符串，用来构成结果 RDD 中的元素。为了让 R 程序能够比较简单地解析输入，我们会把数据以 `mylat, mylon, theirlat, theirlon` 的格式重新组织。这里使用逗号作为分隔符。

例 **6-15**：R 语言的距离程序

```
#!/usr/bin/env Rscript
library("Imap")
f <- file("stdin")
open(f)
while(length(line <- readLines(f,n=1)) > 0) {
  # 处理行
  contents <- Map(as.numeric, strsplit(line, ","))
```

```
  mydist <- gdist(contents[[1]][1], contents[[1]][2],
                  contents[[1]][3], contents[[1]][4],
                  units="m", a=6378137.0, b=6356752.3142, verbose = FALSE)
  write(mydist, stdout())
}
```

如果这段程序写在一个可执行文件 ./src/R/finddistance.R 中，那么使用起来应该像这样：

```
$ ./src/R/finddistance.R
37.75889318222431,-122.42683635321838,37.7614213,-122.4240097
349.2602
coffee
NA
ctrl-d
```

目前为止一切顺利——可以将 `stdin` 中的每一行数据都转为 `stdout` 中的输出了。现在需要做的事情是让每个工作节点都能访问 `finddistance.R`，并调用这个脚本来对 RDD 进行实际的转化操作。这两个任务在 Spark 中都很容易完成，具体可参考例 6-16 至例 6-18。

例 6-16：在 Python 中使用 `pipe()` 调用 `finddistance.R` 的驱动器程序

```
# 使用一个R语言外部程序计算每次呼叫的距离
distScript = "./src/R/finddistance.R"
distScriptName = "finddistance.R"
sc.addFile(distScript)
def hasDistInfo(call):
    """验证一次呼叫是否有计算距离时必需的字段"""
    requiredFields = ["mylat", "mylong", "contactlat", "contactlong"]
    return all(map(lambda f: call[f], requiredFields))
def formatCall(call):
    """将呼叫按新的格式重新组织以使之可以被R程序解析"""
    return "{0},{1},{2},{3}".format(
        call["mylat"], call["mylong"],
        call["contactlat"], call["contactlong"])

pipeInputs = contactsContactList.values().flatMap(
    lambda calls: map(formatCall, filter(hasDistInfo, calls)))
distances = pipeInputs.pipe(SparkFiles.get(distScriptName))
print distances.collect()
```

例 6-17：在 Scala 中使用 `pipe()` 调用

finddistance.R 的驱动器程序

```
// 使用一个R语言外部程序计算每次呼叫的距离
// 将脚本添加到各个节点需要在本次作业中下载的文件的列表中
val distScript = "./src/R/finddistance.R"
val distScriptName = "finddistance.R"
sc.addFile(distScript)
val distances = contactsContactLists.values.flatMap(x => x.map(y =>
  s"$y.contactlay,$y.contactlong,$y.mylat,$y.mylong")).pipe(Seq(
    SparkFiles.get(distScriptName)))
println(distances.collect().toList)
```

例 **6-18**：在 Java 中使用 `pipe()` 调用 `finddistance.R` 的驱动器程序

```
// 使用一个R语言外部程序计算每次呼叫的距离
// 将脚本添加到各个节点需要在本次作业中下载的文件的列表中
String distScript = "./src/R/finddistance.R";
String distScriptName = "finddistance.R";
sc.addFile(distScript);
JavaRDD<String> pipeInputs = contactsContactLists.values()
  .map(new VerifyCallLogs()).flatMap(
  new FlatMapFunction<CallLog[], String>() {
    public Iterable<String> call(CallLog[] calls) {
      ArrayList<String> latLons = new ArrayList<String>();
      for (CallLog call: calls) {
        latLons.add(call.mylat + "," + call.mylong +
                    "," + call.contactlat + "," + call.contactlong);
      }
      return latLons;
    }
  });
JavaRDD<String> distances = pipeInputs.pipe(SparkFiles.get(distScriptName)
System.out.println(StringUtils.join(distances.collect(), ","));
```

通过 `SparkContext.addFile(path)`，可以构建一个文件列表，让每个工作节点在 Spark 作业中下载列表中的文件。这些文件可以来自驱动器的本地文件系统（如前面几个例子中所示范的那样），或者来自 HDFS 或其他 Hadoop 所支持的文件系统，又或者是 HTTP、HTTPS 或 FTP 的 URI 地址。当作业中的行动操作被触发时，这些文件就会被各节点下载，然后我们就可以在工作节点上通过 `SparkFiles.getRootDirectory` 找到它们。我们也可以使用 `SparkFiles.get(Filename)` 来定位单个文件。当然，这只是确保 `pipe()` 能够在各工作节点上找到这个脚本的方法之一。你可以使用其他的远程复制工具来把脚本文件放到各节点可以找到的位置

上。

 所有通过 SparkContext.addFile(path) 添加的文件都存储在同一个目录中，所以有必要使用唯一的名字。

一旦脚本可以访问，RDD 的 pipe() 方法就可以让 RDD 中的元素很容易地通过脚本管道。假设有一个更好版本的 findDistance，可以以命令行参数的形式接收指定的 SEPARATOR。这样的话，下面的两种方法都可以完成工作，不过我们倾向于使用第一种。

- rdd.pipe(Seq(SparkFiles.get("finddistance.R"), ","))

- rdd.pipe(SparkFiles.get("finddistance.R") + " ,")

在第一种方法中，我们将命令调用以可定位的参数序列的形式传递（命令本身在零偏移位置）；而在第二种方法中，我们将它作为一个命令字符串传递，然后 Spark 会将这个字符串拆解为可定位的参数序列。

如果需要的话，也可以通过 pipe() 指定命令行环境变量。只需要把环境变量到对应值的映射表作为 pipe() 的第二个参数传进去，Spark 就会设置好这些值。

你现在至少应该理解了如何使用 pipe()、通过外部命令处理 RDD 中的元素，以及如何将这样的命令脚本分发到集群各节点，并能让工作节点找到这些脚本。

6.6　数值RDD的操作

Spark 对包含数值数据的 RDD 提供了一些描述性的统计操作。这是我们会在第 11 章介绍的更复杂的统计方法和机器学习方法的一个补充。

Spark 的数值操作是通过流式算法实现的，允许以每次一个元素的方式构建出模型。这些统计数据都会在调用 stats() 时通过一次遍历数据计算出来，并以 StatsCounter 对象返回。表 6-2 列出了 StatsCounter 上的可用方法。

表6-2：StatsCounter中可用的汇总统计数据

方法	含义
count()	RDD 中的元素个数
mean()	元素的平均值
sum()	总和
max()	最大值
min()	最小值
variance()	元素的方差
sampleVariance()	从采样中计算出的方差
stdev()	标准差
sampleStdev()	采样的标准差

如果你只想计算这些统计数据中的一个，也可以直接对 RDD 调用对应的方法，比如 `rdd.mean()` 或者 `rdd.sum()`。

在例 6-19 至例 6-21 中，我们会使用汇总统计来从数据中移除一些异常值。由于我们会两次使用同一个 RDD（一次用来计算汇总统计数据，另一次用来移除异常值），因此应该把这个 RDD 缓存下来。回到呼叫日志的示例中，来看看如何从呼叫日志中移除距离过远的联系点。

例 6-19：用 Python 移除异常值

```
# 要把String类型RDD转为数字数据，这样才能
# 使用统计函数并移除异常值
distanceNumerics = distances.map(lambda string: float(string))
stats = distanceNumerics.stats()
stddev = std.stdev()
mean = stats.mean()
reasonableDistances = distanceNumerics.filter(
  lambda x: math.fabs(x - mean) < 3 * stddev)
print reasonableDistances.collect()
```

例 6-20：用 Scala 移除异常值

```
// 现在要移除一些异常值，因为有些地点可能是误报的
// 首先要获取字符串RDD并将它转换为双精度浮点型
val distanceDouble = distance.map(string => string.toDouble)
val stats = distanceDoubles.stats()
val stddev = stats.stdev
val mean = stats.mean
val reasonableDistances = distanceDoubles.filter(x => math.abs(x-mean) < 3
println(reasonableDistance.collect().toList)
```

例 **6-21**：用 Java 移除异常值

```
// 首先要把String类型RDD转为DoubleRDD，这样才能使用统计函数
JavaDoubleRDD distanceDoubles = distances.mapToDouble(new DoubleFunction<S
    public double call(String value) {
      return Double.parseDouble(value);
    }});
final StatCounter stats = distanceDoubles.stats();
final Double stddev = stats.stdev();
final Double mean = stats.mean();
JavaDoubleRDD reasonableDistances =
  distanceDoubles.filter(new Function<Double, Boolean>() {
    public Boolean call(Double x) {
      return (Math.abs(x-mean) < 3 * stddev);}});
System.out.println(StringUtils.join(reasonableDistance.collect(), ","));
```

至此，这个示例应用已经全部讲完。在这个应用中，我们使用了累加器、广播变量、基于分区处理、外部程序接口调用以及汇总统计。完整的源代码分别可以从 src/python/ChapterSixExample.py、src/main/scala/com/oreilly/learningsparkexamples/scala/ChapterSixExample.scala 以及 src/main/java/com/oreilly/learningsparkexamples/java/ChapterSixExample.java 中找到。

6.7 总结

在本章中，我们介绍了 Spark 编程中的一些进阶特性，你可以使用这些特性使让你的程序变得更高效、更强大。后续章节会介绍部署与调优 Spark 应用，以及 Spark 内建的 SQL 库、流计算库和机器学习库。此外，我们也会开始接触一些更复杂更完整的示例程序，这些示例会用到目前为止所学过的许多功能，并且可以为你自己的 Spark 应用带来指导和启发。

第 7 章　在集群上运行 Spark

7.1　简介

到目前为止，本书一直在讲如何利用 Spark shell 学习 Spark，示例程序也都运行在 Spark 本地模式下。而 Spark 的一大好处就是可以通过增加机器数量并使用集群模式运行，来扩展程序的计算能力。好在编写用于在集群上并行执行的 Spark 应用所使用的 API 跟本书之前几章所讨论的完全一样。也就是说，你可以在小数据集上利用本地模式快速开发并验证你的应用，然后无需修改代码就可以在大规模集群上运行。

本章首先介绍分布式 Spark 应用的运行环境架构，然后讨论在集群上运行 Spark 应用时的一些配置项。Spark 可以在各种各样的集群管理器（Hadoop YARN、Apache Mesos，还有 Spark 自带的独立集群管理器）上运行，所以 Spark 应用既能够适应专用集群，又能用于共享的云计算环境。我们会对各种使用情况下的优缺点和配置方法进行探讨。同时，也会讨论 Spark 应用在调度、部署、配置等各方面的一些细节。读完本章之后，你应该能学会运行分布式 Spark 程序的一切技能。下一章会介绍如何对 Spark 应用进行调试和性能调优。

7.2　Spark运行时架构

在深入探讨如何在集群上运行 spark 之前，先来了解一下 Spark 在分布式环境中的架构（见图 7-1），这有助于理解在集群上运行 Spark 的一些具体细节。

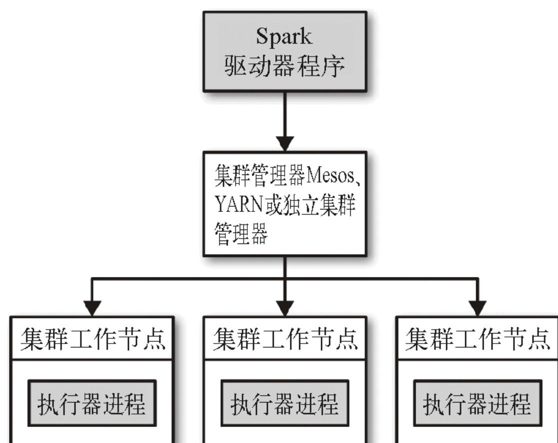


图 **7-1**：分布式 **Spark** 应用中的组件

在分布式环境下，Spark 集群采用的是主 / 从结构。在一个 Spark 集群中，有一个节点负责中央协调，调度各个分布式工作节点。这个中央协调节点被称为驱动器（Driver）节点，与之对应的工作节点被称为执行器（executor）节点。驱动器节点可以和大量的执行器节点进行通信，它们也都作为独立的 Java 进程运行。驱动器节点和所有的执行器节点一起被称为一个 **Spark** 应用（application）。

Spark 应用通过一个叫作集群管理器（Cluster Manager）的外部服务在集群中的机器上启动。Spark 自带的集群管理器被称为独立集群管理器。Spark 也能运行在 Hadoop YARN 和 Apache Mesos 这两大开源集群管理器上。

7.2.1　驱动器节点

Spark 驱动器是执行你的程序中的 `main()` 方法的进程。它执行用户编写的用来创建 SparkContext、创建 RDD，以及进行 RDD 的转化操作和行动操作的代码。其实，当你启动 Spark shell 时，你就启动了一个 Spark 驱动器程序（相信你还记得，Spark shell 总是会预先加载一个叫作 `sc` 的 SparkContext 对象）。驱动器程序一旦终止，Spark 应用也就结束了。

驱动器程序在 Spark 应用中有下述两个职责。

- 把用户程序转为任务

Spark 驱动器程序负责把用户程序转为多个物理执行的单元，这些单元也被称为任务（task）。从上层来看，所有的 Spark 程序都遵循同样的结构：程序从输入数据创建一系列 RDD，再使用转化操作派生出新的 RDD，最后使用行动操作收集或存储结果 RDD 中的数据。Spark 程序其实是隐式地创建出了一个由操作组成的逻辑上的有向无环图（Directed Acyclic Graph，简称 DAG）。当驱动器程序运行时，它会把这个逻辑图转为物理执行计划。

Spark 会对逻辑执行计划作一些优化，比如将连续的映射转为流水线化执行，将多个操作合并到一个步骤中等。这样 Spark 就把逻辑计划转为一系列步骤（stage）。而每个步骤又由多个任务组成。这些任务会被打包并送到集群中。任务是 Spark 中最小的工作单元，用户程序通常要启动成百上千的独立任务。

- 为执行器节点调度任务

  有了物理执行计划之后，Spark 驱动器程序必须在各执行器进程间协调任务的调度。执行器进程启动后，会向驱动器进程注册自己。因此，驱动器进程始终对应用中所有的执行器节点有完整的记录。每个执行器节点代表一个能够处理任务和存储 RDD 数据的进程。

  Spark 驱动器程序会根据当前的执行器节点集合，尝试把所有任务基于数据所在位置分配给合适的执行器进程。当任务执行时，执行器进程会把缓存数据存储起来，而驱动器进程同样会跟踪这些缓存数据的位置，并且利用这些位置信息来调度以后的任务，以尽量减少数据的网络传输。

  驱动器程序会将一些 Spark 应用的运行时的信息通过网页界面呈现出来，默认在端口 4040 上。比如，在本地模式下，访问 http://localhost:4040 就可以看到这个网页了。我们会在第 8 章更加详细地讲解 Spark 的网页用户界面以及 Spark 的作业调度机制。

7.2.2　执行器节点

Spark 执行器节点是一种工作进程，负责在 Spark 作业中运行任务，任务间相互独立。Spark 应用启动时，执行器节点就被同时启动，并且始终伴随着整个 Spark 应用的生命周期而存在。如果有执行器节点发生了异常或崩溃，Spark 应用也可以继续执行。执行器进程有两大

作用：第一，它们负责运行组成 Spark 应用的任务，并将结果返回给驱动器进程；第二，它们通过自身的块管理器（Block Manager）为用户程序中要求缓存的 RDD 提供内存式存储。RDD 是直接缓存在执行器进程内的，因此任务可以在运行时充分利用缓存数据加速运算。



本地模式中的驱动器程序和执行器程序

在本书中，大部分示例都是在本地模式运行 Spark。在本地模式下，Spark 驱动器程序和各执行器程序在同一个 Java 进程中运行。这是一个特例；执行器程序通常都运行在专用的进程中。

7.2.3　集群管理器

到目前为止，我们已经介绍了驱动器节点和执行器节点的抽象概念。那么，驱动器节点和执行器节点是如何启动的呢？ Spark 依赖于集群管理器来启动执行器节点，而在某些特殊情况下，也依赖集群管理器来启动驱动器节点。集群管理器是 Spark 中的可插拔式组件。这样，除了 Spark 自带的独立集群管理器，Spark 也可以运行在其他外部集群管理器上，比如 YARN 和 Mesos。



Spark 文档中始终使用驱动器节点和执行器节点的概念来描述执行 Spark 应用的进程。而主节点（master）和工作节点（worker）的概念则被用来分别表述集群管理器中的中心化的部分和分布式的部分。这些概念很容易混淆，所以要格外小心。例如，Hadoop YARN 会启动一个叫作资源管理器（Resource Manager）的主节点守护进程，以及一系列叫作节点管理器（Node Manager）的工作节点守护进程。而在 YARN 的工作节点上，Spark 不仅可以运行执行器进程，还可以运行驱动器进程。

7.2.4　启动一个程序

不论你使用的是哪一种集群管理器，你都可以使用 Spark 提供的统一脚本 `spark-submit` 将你的应用提交到那种集群管理器上。通过不同的配置选项，`spark-submit` 可以连接到相应的集群管理器上，并控制应用所使用的资源数量。在使用某些特定集群管理器时，`spark-submit` 也可以将驱动器节点运行在集群内部（比如一个

YARN 的工作节点）。但对于其他的集群管理器，驱动器节点只能被运行在本地机器上。我们会在下一节更加详细地讲解 `spark-submit`。

7.2.5　小结

回顾在集群上运行 Spark 应用的详细过程，可把本节的主要概念作如下总结。

(1) 用户通过 `spark-submit` 脚本提交应用。

(2) `spark-submit` 脚本启动驱动器程序，调用用户定义的 `main()` 方法。

(3) 驱动器程序与集群管理器通信，申请资源以启动执行器节点。

(4) 集群管理器为驱动器程序启动执行器节点。

(5) 驱动器进程执行用户应用中的操作。根据程序中所定义的对 RDD 的转化操作和行动操作，驱动器节点把工作以任务的形式发送到执行器进程。

(6) 任务在执行器程序中进行计算并保存结果。

(7) 如果驱动器程序的 `main()` 方法退出，或者调用了 `SparkContext.stop()`，驱动器程序会终止执行器进程，并且通过集群管理器释放资源。

7.3　使用spark-submit部署应用

前面学习过，Spark 为各种集群管理器提供了统一的工具来提交作业，这个工具是 `spark-submit`。第 2 章中有一个使用 `spark-submit` 提交 Python 程序的简单示例，这里在例 7-1 中重复一遍。

例 **7-1**：提交 Python 应用

```
bin/spark-submit my_script.py
```

如果在调用 `spark-submit` 时除了脚本或 JAR 包的名字之外没有别的参数，那么这个 Spark 程序只会在本地执行。当我们希望将应用

提交到 Spark 独立集群上的时候，可以将独立集群的地址和希望启动的每个执行器进程的大小作为附加标记提供，如例 7-2 所示。

例 **7-2**：提交应用时添加附加参数

```
bin/spark-submit --master spark://host:7077 --executor-memory 10g my_script.py
```

`--master` 标记指定要连接的集群 URL；在这个示例中，spark:// 表示集群使用独立模式（见表 7-1）。稍后会讨论其他的 URL 类型。

表**7-1**：**spark-submit**的**--master**标记可以接收的值

描述
连接到指定端口的 Spark 独立集群上。默认情况下 Spark 独立主节点使用 7077 端口
连接到指定端口的 Mesos 集群上。默认情况下 Mesos 主节点监听 5050 端口 yarn 连接到一个 YARN 集群。当在 YARN 上运行时，需要设置环境变量 HADOOP_CONF_DIR 指向 Hadoop 配置目录，以获取集群信息
运行本地模式，使用单核
运行本地模式，使用 N 个核心
运行本地模式，使用尽可能多的核心

除了集群 URL，spark-submit 还提供了各种选项，可以让你控制应用每次运行的各项细节。这些选项主要分为两类。第一类是调度信息，比如你希望为作业申请的资源量（如例 7-2 所示）。第二类是应用的运行时依赖，比如需要部署到所有工作节点上的库和文件。

spark-submit 的一般格式见例 7-3。

例 **7-3**：spark-submit 的一般格式

```
bin/spark-submit [options] <app jar | python file> [app options]
```

[options] 是要传给 spark-submit 的标记列表。你可以运行 spark-submit --help 列出所有可以接收的标记。表 7-2 列出了一些常见的标记。

<app jar | python File> 表示包含应用入口的 JAR 包或 Python 脚本。

[app options] 是传给你的应用的选项。如果你的程序要处理传给

`main()` 方法的参数，它只会得到 `[app options]` 对应的标记，不会得到 `spark-submit` 的标记。

表7-2：**spark-submit**的一些常见标记

标记	描述
`--master`	表示要连接的集群管理器。这个标记可接收的选项见表 7-1
`--deploy-mode`	选择在本地（客户端“client”）启动驱动器程序，还是在集群中的一台工作节点机器（集群“cluster”）上启动。在客户端模式下，`spark-submit` 会将驱动器程序运行在 `spark-submit` 被调用的这台机器上。在集群模式下，驱动器程序会被传输并执行于集群的一个工作节点上。默认是本地模式
`--class`	运行 Java 或 Scala 程序时应用的主类
`--name`	应用的显示名，会显示在 Spark 的网页用户界面中
`--jars`	需要上传并放到应用的 CLASSPATH 中的 JAR 包的列表。如果应用依赖于少量第三方的 JAR 包，可以把它们放在这个参数里
`--files`	需要放到应用工作目录中的文件的列表。这个参数一般用来放需要分发到各节点的数据文件
`--py-files`	需要添加到 PYTHONPATH 中的文件的列表。其中可以包含 .py、.egg 以及 .zip 文件
`--executor-memory`	执行器进程使用的内存量，以字节为单位。可以使用后缀指定更大的单位，比如“512m”（512 MB）或“15g”（15 GB）
`--driver-memory`	驱动器进程使用的内存量，以字节为单位。可以使用后缀指定更大的单位，比如“512m”（512 MB）或“15g”（15 GB）

`spark-submit` 还允许通过 `--conf prop=value` 标记设置任意的 SparkConf 配置选项，也可以使用 `--properties-File` 指定一个包含键值对的属性文件。第 8 章会讨论 Spark 的配置系统。

例 7-4 展示了一些使用各种选项调用 `spark-submit` 的例子，这些调用语句都比较长。

例 7-4：使用各种选项调用 `spark-submit`

```
# 使用独立集群模式提交Java应用
$ ./bin/spark-submit \
  --master spark://hostname:7077 \
  --deploy-mode cluster \
  --class com.databricks.examples.SparkExample \
  --name "Example Program" \
  --jars dep1.jar,dep2.jar,dep3.jar \
  --total-executor-cores 300 \
  --executor-memory 10g \
  myApp.jar "options" "to your application" "go here"

# 使用YARN客户端模式提交Python应用
$ export HADOP_CONF_DIR=/opt/hadoop/conf
$ ./bin/spark-submit \
  --master yarn \
```

```
--py-files somelib-1.2.egg,otherlib-4.4.zip,other-file.py \
--deploy-mode client \
--name "Example Program" \
--queue exampleQueue \
--num-executors 40 \
--executor-memory 10g \
my_script.py "options" "to your application" "go here"
```

7.4　打包代码与依赖

本书中大部分示例程序都是独立的，不依赖于 Spark 以外的库。而通常用户程序则需要依赖第三方的库。如果你的程序引入了任何既不在 `org.apache.spark` 包内也不属于语言运行时的库的依赖，你就需要确保所有的依赖在该 Spark 应用运行时都能被找到。

对于 Python 用户而言，有多种安装第三方库的方法。由于 PySpark 使用工作节点机器上已有的 Python 环境，你可以通过标准的 Python 包管理器（比如 `pip` 和 `easy_install`）直接在集群中的所有机器上安装所依赖的库，或者把依赖手动安装到 Python 安装目录下的 site-packages/ 目录中。你也可以使用 `spark-submit` 的 `--py-Files` 参数提交独立的库，这样它们也会被添加到 Python 解释器的路径中。如果你没有在集群上安装包的权限，可以手动添加依赖库，这也很方便，但是要防范与已经安装在集群上的那些包发生冲突。



关于 **Spark** 本身

当你提交应用时，绝不要把 Spark 本身放在提交的依赖中。`spark-submit` 会自动确保 Spark 在你的程序的运行路径中。

Java 和 Scala 用户也可以通过 `spark-submit` 的 `--jars` 标记提交独立的 JAR 包依赖。当只有一两个库的简单依赖，并且这些库本身不依赖于其他库时，这种方法比较合适。但是一般 Java 和 Scala 的工程会依赖很多库。当你向 Spark 提交应用时，你必须把应用的整个依赖传递图中的所有依赖都传给集群。你不仅要传递你直接依赖的库，还要传递这些库的依赖，以及它们的依赖的依赖，等等。手动维护和提交全部的依赖 JAR 包是很笨的方法。事实上，常规的做法是使用构建工具，生成单个大 JAR 包，包含应用的所有的传递依赖。这通常被称为超级（uber）JAR 或者组合（assembly）JAR，大多数

Java 或 Scala 的构建工具都支持生成这样的工件。

Java 和 Scala 中使用最广泛的构建工具是 Maven 和 sbt。它们都可以用于这两种语言，不过 Maven 通常用于 Java 工程，而 sbt 则一般用于 Scala 工程。在这里，我们会分别针对使用这两种工具构建 Spark 应用给出相应的例子。你可以把这里的示例当作自己构建 Spark 工程的模板。

7.4.1　使用Maven构建的用Java编写的Spark应用

下面来看一个有多个依赖的 Java 工程，在示例中我们会将它打包为一个超级 JAR 包。例 7-5 提供了 Maven 的 pom.xml 文件，其中包含本次构建的定义。这个例子没有展示实际的 Java 代码与工程的目录结构。Maven 中默认的用户代码在工程根目录（该目录应包含 pom.xml 文件）下的 src/main/java 目录中。

例 **7-5**：使用 Maven 构建的 Spark 应用的 pom.xml 文件

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <!-- 工程相关信息 -->
  <groupId>com.databricks</groupId>
  <artifactId>example-build</artifactId>
  <name>Simple Project</name>
  <packaging>jar</packaging>
  <version>1.0</version>

  <dependencies>
    <!-- Spark依赖 -->
    <dependency>
      <groupId>org.apache.spark</groupId>
      <artifactId>spark-core_2.10</artifactId>
      <version>1.2.0</version>
      <scope>provided</scope>
    </dependency>
    <!-- 第三方库 -->
    <dependency>
      <groupId>net.sf.jopt-simple</groupId>
      <artifactId>jopt-simple</artifactId>
      <version>4.3</version>
    </dependency>
    <!-- 第三方库 -->
    <dependency>
      <groupId>joda-time</groupId>
      <artifactId>joda-time</artifactId>
```

```
      <version>2.0</version>
    </dependency>
  </dependencies>

  <build>
    <plugins>
      <!-- 用来创建超级JAR包的Maven shade插件 -->
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.3</version>
        <executions>
          <execution>
            <phase>package</phase>
            <goals>
              <goal>shade</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

这个工程声明了两个传递依赖：jopt-simple 和 joda-time，前者用来作选项解析，而后者是一个用来处理时间日期转换的工具库。这个工程也依赖于 Spark，不过我们把 Spark 标记为 provided 来确保 Spark 不与应用依赖的其他工件打包在一起。构建时，我们使用 maven-shade-plugin 插件来创建出包含所有依赖的超级 JAR 包。你可以让 Maven 在每次进行打包时执行插件的 shade 目标来使用此插件。使用这样的构建配置，超级 JAR 包就会在执行 mvn package 时自动创建出来（见例 7-6）。

例 **7-6**：打包使用 Maven 构建的 Spark 应用

```
$ mvn package
# 在目标路径中，可以看到超级JAR包和原版打包方法生成的JAR包
$ ls target/
example-build-1.0.jar
original-example-build-1.0.jar
# 展开超级JAR包可以看到依赖库中的类
$ jar tf target/example-build-1.0.jar
...
joptsimple/HelpFormatter.class
...
org/joda/time/tz/UTCProvider.class
...
# 超级JAR可以直接传给spark-submit
```

```
$ /path/to/spark/bin/spark-submit --master local ... target/example-build-
```

7.4.2　使用sbt构建的用Scala编写的Spark应用

sbt 是一个通常在 Scala 工程中使用的比较新的构建工具。sbt 预期的目录结构和 Maven 相似。在工程的根目录中，你要创建出一个叫作 build.sbt 的构建文件，源代码则应该放在 src/main/scala 中。sbt 构建文件是用配置语言写成的，在这个文件中我们把值赋给特定的键，用来定义工程的构建。例如，有一个键叫作 `name`，是用来指定工程名字的，还有一个键叫作 `libraryDependencies`，用来指定工程的依赖列表。例 7-7 给出了一个依赖于 Spark 和一些第三方库的简单应用的 sbt 完整构建文件。这个构建文件适用于 sbt 0.13。sbt 正在快速发展中，因此你可能需要读一读最新的文档，以了解构建文件格式上的改变。

例 **7-7**：使用 sbt 0.13 的 Spark 应用的 build.sbt 文件

```
import AssemblyKeys._

name := "Simple Project"

version := "1.0"

organization := "com.databricks"

scalaVersion := "2.10.3"

libraryDependencies ++= Seq(
    // Spark依赖
    "org.apache.spark" % "spark-core_2.10" % "1.2.0" % "provided",
    // 第三方库
    "net.sf.jopt-simple" % "jopt-simple" % "4.3",
    "joda-time" % "joda-time" % "2.0"
)

// 这条语句打开了assembly插件的功能
assemblySettings

// 配置assembly插件所使用的JAR
jarName in assembly := "my-project-assembly.jar"

// 一个用来把Scala本身排除在组合JAR包之外的特殊选项，因为Spark
// 已经包含了Scala
assemblyOption in assembly :=
  (assemblyOption in assembly).value.copy(includeScala = false)
```

这个构建文件的第一行从插件中引入了一些功能，这个插件用来支持创建项目的组合 JAR 包。要使用这个插件，需要在 project/ 目录下加入一个小文件，来列出对插件的依赖。我们只需要创建出 project/assembly.sbt 文件，并在其中加入 `addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")`。`sbt-assembly` 的实际版本可能会因使用的 sbt 版本不同而变化。例 7-8 适用于 sbt 0.13。

例 **7-8**：在 sbt 工程构建中添加 `assembly` 插件

```
# 显示project/assembly.sbt的内容
$ cat project/assembly.sbt
addSbtPlugin("com.eed3si9n" % "sbt-assembly" % "0.11.2")
```

定义好了构建文件之后，就可以创建出一个完全组合打包的 Spark 应用 JAR 包（例 7-9）。

例 **7-9**：打包使用 sbt 构建的 Spark 应用

```
$ sbt assembly
# 在目标路径中，可以看到一个组合JAR包
$ ls target/scala-2.10/
my-project-assembly.jar
# 展开组合JAR包可以看到依赖库中的类
$ jar tf target/scala-2.10/my-project-assembly.jar
...
joptsimple/HelpFormatter.class
...
org/joda/time/tz/UTCProvider.class
...
# 组合JAR可以直接传给spark-submit
$ /path/to/spark/bin/spark-submit --master local ...
  target/scala-2.10/my-project-assembly.jar
```

7.4.3　依赖冲突

当用户应用与 Spark 本身依赖同一个库时可能会发生依赖冲突，导致程序崩溃。这种情况不是很常见，但是出现的时候也让人很头疼。通常，依赖冲突表现为 Spark 作业执行过程中抛出 `NoSuchMethodError`、`ClassNotFoundException`，或其他与类加载相关的 JVM 异常。对于这种问题，主要有两种解决方式：一是修改你的应用，使其使用的依赖库的版本与 Spark 所使用的相同，二是使用通常被称为“shading”的方式打包你的应用。Maven 构建工

具通过对例 7-5 中使用的插件（事实上，shading 的功能也是这个插件取名为 `maven-shade-plugin` 的原因）进行高级配置来支持这种打包方式。shading 可以让你以另一个命名空间保留冲突的包，并自动重写应用的代码使得它们使用重命名后的版本。这种技术有些简单粗暴，不过对于解决运行时依赖冲突的问题非常有效。如果要了解使用这种打包方式的具体步骤，请参阅你所使用的构建工具对应的文档。

7.5　Spark应用内与应用间调度

刚才的例子中只有一个用户向集群提交作业。而在现实中，许多集群是在多个用户间共享的。共享的环境会带来调度方面的挑战：如果两个用户都启动了希望使用整个集群所有资源的应用，该如何处理呢？Spark 有一系列调度策略来保障资源不会被过度使用，还允许工作负载设置优先级。

在调度多用户集群时，Spark 主要依赖集群管理器来在 Spark 应用间共享资源。当 Spark 应用向集群管理器申请执行器节点时，应用收到的执行器节点个数可能比它申请的更多或者更少，这取决于集群的可用性与争用。许多集群管理器支持队列，可以为队列定义不同优先级或容量限制，这样 Spark 就可以把作业提交到相应的队列中。请查看你所使用的集群管理器的文档获取详细信息。

Spark 应用有一种特殊情况，就是那些长期运行（long lived）的应用。这意味着这些应用从不主动退出。Spark SQL 中的 JDBC 服务器就是一个长期运行的 Spark 应用。当 JDBC 服务器启动后，它会从集群管理器获得一系列执行器节点，然后就成为用户提交 SQL 查询的永久入口。由于这个应用本身就是为多用户调度工作的，所以它需要一种细粒度的调度机制来强制共享资源。Spark 提供了一种用来配置应用内调度策略的机制。Spark 内部的公平调度器（Fair Scheduler）会让长期运行的应用定义调度任务的优先级队列。本书对这部分内容未作深入探讨，若想了解详情，请参考公平调度器的官方文档：（http://spark.apache.org/docs/latest/job-scheduling.html）。

7.6　集群管理器

Spark 可以运行在各种集群管理器上，并通过集群管理器访问集群中的机器。如果你只想在一堆机器上运行 Spark，那么自带的独立模式是部署该集群最简单的方法。然而，如果你有一个需要与别的分布式

应用共享的集群（比如既可以运行 Spark 作业又可以运行 Hadoop MapReduce 作业），Spark 也可以运行在两个广泛使用的集群管理器——Hadoop YARN 与 Apache Mesos 上面。最后，在把 Spark 部署到 Amazon EC2 上时，Spark 有个自带的脚本可以启动独立模式集群以及各种相关服务。本节会介绍如何在这些环境中分别运行 Spark。

7.6.1 独立集群管理器

Spark 独立集群管理器提供在集群上运行应用的简单方法。这种集群管理器由一个主节点和几个工作节点组成，各自都分配有一定量的内存和 CPU 核心。当提交应用时，你可以配置执行器进程使用的内存量，以及所有执行器进程使用的 CPU 核心总数。

1. 启动独立集群管理器

要启动独立集群管理器，你既可以通过手动启动一个主节点和多个工作节点来实现，也可以使用 Spark 的 sbin 目录中的启动脚本来实现。启动脚本使用最简单的配置选项，但是需要预先设置机器间的 SSH 无密码访问。在 Spark 1.1 中，启动脚本仅适用于 Mac OS X 和 Linux。我们会先讲这两个操作系统上独立集群管理器的启动方法，然后再展示在其他平台上如何启动集群。

要使用集群启动脚本，请按如下步骤执行。

(1) 将编译好的 Spark 复制到所有机器的一个相同的目录下，比如 /home/yourname/spark。

(2) 设置好从主节点机器到其他机器的 SSH 无密码登陆。这需要在所有机器上有相同的用户账号，并在主节点上通过 `ssh-keygen` 生成 SSH 私钥，然后将这个私钥放到所有工作节点的 .ssh/authorized_keys 文件中。如果你之前没有设置过这种配置，你可以使用如下命令：

```
# 在主节点上：运行ssh-keygen并接受默认选项
$ ssh-keygen -t dsa
Enter file in which to save the key (/home/you/.ssh/id_dsa): [回车]
Enter passphrase (empty for no passphrase): [空]
Enter same passphrase again: [空]

# 在工作节点上：
# 把主节点的~/.ssh/id_dsa.pub文件复制到工作节点上，然后使用：
$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
$ chmod 644 ~/.ssh/authorized_keys
```

(3) 编辑主节点的 conf/slaves 文件并填上所有工作节点的主机名。

(4) 在主节点上运行 `sbin/start-all.sh`（要在主节点上运行而不是在工作节点上）以启动集群。如果全部启动成功，你不会得到需要密码的提示符，而且可以在 http://masternode:8080 看到集群管理器的网页用户界面，上面显示着所有的工作节点。

(5) 要停止集群，在主节点上运行 `bin/stop-all.sh`。

如果你使用的不是 UNIX 系统，或者想手动启动集群，你也可以使用 Spark 的 bin/ 目录下的 `spark-class` 脚本分别手动启动主节点和工作节点。在主节点上，输入：

```
bin/spark-class org.apache.spark.deploy.master.Master
```

然后在工作节点上输入：

```
bin/spark-class org.apache.spark.deploy.worker.Worker spark://masternode:
```

（其中 `masternode` 是你的主节点的主机名）。在 Windows 中，使用 \ 来代替 /。

默认情况下，集群管理器会选择合适的默认值自动为所有工作节点分配 CPU 核心与内存。配置独立集群管理器的更多细节请参考 Spark 的官方文档（http://spark.apache.org/docs/latest/spark-standalone.html）。

2. 提交应用

要向独立集群管理器提交应用，需要把 `spark://masternode:7077` 作为主节点参数传给 `spark-submit`。例如：

```
spark-submit --master spark://masternode:7077 yourapp
```

这个集群的 URL 也显示在独立集群管理器的网页界面（位于 http://masternode:8080）上。注意，提交时使用的主机名和端口号必须精确匹配用户界面中的 URL。这有可能会使得使用 IP 地址而非主机名

的用户遇到问题。即使 IP 地址绑定到同一台主机上，如果名字不是完全匹配的话，提交也会失败。有些管理员可能会配置 Spark 不使用 7077 端口而使用别的端口。要确保主机名和端口号的一致性，一个简单的方法是从主节点的用户界面中直接复制粘贴 URL。

你可以使用 `--master` 参数以同样的方式启动 `spark-shell` 或 `pyspark`，来连接到该集群上：

```
spark-shell --master spark://masternode:7077
pyspark --master spark://masternode:7077
```

要检查你的应用或者 shell 是否正在运行，你需要查看集群管理器的网页用户界面 http://masternode:8080 并确保：（1）应用连接上了（即出现在了 Running Applications 中）；（2）列出的所使用的核心和内存均大于 0。

 阻碍应用运行的一个常见陷阱是为执行器进程申请的内存（`spark-submit` 的 `--executor-memory` 标记传递的值）超过了集群所能提供的内存总量。在这种情况下，独立集群管理器始终无法为应用分配执行器节点。请确保应用申请的值能够被集群接受。

最后，独立集群管理器支持两种部署模式。在这两种模式中，应用的驱动器程序运行在不同的地方。在客户端模式中（默认情况），驱动器程序会运行在你执行 `spark-submit` 的机器上，是 `spark-submit` 命令的一部分。这意味着你可以直接看到驱动器程序的输出，也可以直接输入数据进去（通过交互式 shell），但是这要求你提交应用的机器与工作节点间有很快的网络速度，并且在程序运行的过程中始终可用。相反，在集群模式下，驱动器程序会作为某个工作节点上一个独立的进程运行在独立集群管理器内部。它也会连接主节点来申请执行器节点。在这种模式下，`spark-submit` 是“一劳永逸”型，你可以在应用运行时关掉你的电脑。你还可以通过集群管理器的网页用户界面访问应用的日志。向 `spark-submit` 传递 `--deploy-mode cluster` 参数可以切换到集群模式。

3. 配置资源用量

如果在多应用间共享 Spark 集群，你需要决定如何在执行器进程间分配资源。独立集群管理器使用基础的调度策略，这种策略允许限制各

个应用的用量来让多个应用并发执行。Apache Mesos 支持应用运行时的更动态的资源共享，而 YARN 则有分级队列的概念，可以让你限制不同类别的应用的用量。

在独立集群管理器中，资源分配靠下面两个设置来控制。

- 执行器进程内存

  你可以通过 `spark-submit` 的 `--executor-memory` 参数来配置此项。每个应用在每个工作节点上最多拥有一个执行器进程 1，因此这个设置项能够控制执行器节点占用工作节点的多少内存。此设置项的默认值是 1 GB，在大多数服务器上，你可能需要提高这个值来充分利用机器。

- 占用核心总数的最大值

  这是一个应用中所有执行器进程所占用的核心总数。此项的默认值是无限。也就是说，应用可以在集群所有可用的节点上启动执行器进程。对于多用户的工作负载来说，你应该要求用户限制他们的用量。你可以通过 `spark-submit` 的 `--total-executorcores` 参数设置这个值，或者在你的 Spark 配置文件中设置 `spark.cores.max` 的值。

> 1虽然一个从节点只能运行一个执行器进程，但是一台机器上可以运行多个从节点。——译者注

要验证这些设定，你可以从独立集群管理器的网页用户界面（http://masternode:8080）中查看当前的资源分配情况。

最后，独立集群管理器在默认情况下会为每个应用使用尽可能分散的执行器进程。例如，假设你有一个 20 个物理节点的集群，每个物理节点是一个四核的机器，然后你使用 `--executor-memory 1G` 和 `--total-executor-cores 8` 提交应用。这样 Spark 就会在不同机器上启动 8 个执行器进程，每个 1 GB 内存。Spark 默认这样做，以尽量实现对于运行在相同机器上的分布式文件系统（比如 HDFS）的数据本地化，因为这些文件系统通常也把数据分散到所有物理节点上。如果你愿意，可以通过设置配置属性 `spark.deploy.spreadOut` 为 `false` 来要求 Spark 把执行器进程合并到尽量少的工作节点中。在这样的情况下，前面那个应用就只会得到两个执行器节点，每个有 1 GB 内存和 4 个核心。这一设定会影响运行在独立模式集群上的所有应用，并且必须在启动独立集群管

理器之前设置好。

4. 高度可用性

当在生产环境中运行时，你会希望你的独立模式集群始终能够接收新的应用，哪怕当前集群中所有的节点都崩溃了。其实，独立模式能够很好地支持工作节点的故障。如果你想让集群的主节点也拥有高度可用性，Spark 还支持使用 Apache ZooKeeper（一个分布式协调系统）来维护多个备用的主节点，并在一个主节点失败时切换到新的主节点上。为独立集群配置 ZooKeeper 不在本书的探讨范围内，不过在 Spark 官方文档（https://spark.apache.org/docs/latest/spark-standalone.html#high-availability）中有所描述。

7.6.2 Hadoop YARN

YARN 是在 Hadoop 2.0 中引入的集群管理器，它可以让多种数据处理框架运行在一个共享的资源池上，并且通常安装在与 Hadoop 文件系统（简称 HDFS）相同的物理节点上。在这样配置的 YARN 集群上运行 Spark 是很有意义的，它可以让 Spark 在存储数据的物理节点上运行，以快速访问 HDFS 中的数据。

在 Spark 里使用 YARN 很简单：你只需要设置指向你的 Hadoop 配置目录的环境变量，然后使用 `spark-submit` 向一个特殊的主节点 URL 提交作业即可。

第一步是找到你的 Hadoop 的配置目录，并把它设为环境变量 `HADOOP_CONF_DIR`。这个目录包含 yarn-site.xml 和其他配置文件；如果你把 Hadoop 装到 HADOOP_HOME 中，那么这个目录通常位于 HADOOP_HOME/conf 中，否则可能位于系统目录 /etc/hadoop/conf 中。然后用如下方式提交你的应用：

```
export HADOOP_CONF_DIR="..."
spark-submit --master yarn yourapp
```

和独立集群管理器一样，有两种将应用连接到集群的模式：客户端模式以及集群模式。在客户端模式下应用的驱动器程序运行在提交应用的机器上（比如你的笔记本电脑），而在集群模式下，驱动器程序也运行在一个 YARN 容器内部。你可以通过 `spark-submit` 的 `--deploy-mode` 参数设置不同的模式。2

> 2此处为原书错误，自 Spark 1.0.0 起连接到 YARN 集群有两种模式，分别使用 `yarn-client` 和 `yarn-cluster` 两种参数。请参阅 Spark 官方文档。——译者注

Spark 的交互式 shell 以及 `pyspark` 也都可以运行在 YARN 上。只要设置好 `HADOOP_CONF_ DIR` 并对这些应用使用 `--master yarn` 参数即可。注意，由于这些应用需要从用户处获取输 入，所以只能运行于客户端模式下。

配置资源用量

当在 YARN 上运行时，根据你在 `spark-submit` 或 `spark-shell` 等脚本的 `--num-executors` 标记中设置的值，Spark 应用会使用固定数量的执行器节点。默认情况下，这个值仅为 2，所以你可能需要提高它。你也可以设置通过 `--executor-memory` 设置每个执行器的内存用量，通过 `--executor-cores` 设置每个执行器进程从 YARN 中占用的核心数目。对于给定的硬件资源，Spark 通常在用量较大而总数较少的执行器组合（使用多核与更多内存）上表现得更好，因为这样 Spark 可以优化各执行器进程间的通信。然而，需要注意的是，一些集群限制了每个执行器进程的最大内存（默认为 8 GB），不让你使用更大的执行器进程。

出于资源管理的目的，某些 YARN 集群被设置为将应用调度到多个队列中。使用 `--queue` 选项来选择你的队列的名字。

要了解关于 YARN 的更多配置选项的相关信息，可以查阅 Spark 官方文档（http://spark.apache.org/docs/latest/submitting-applications.html）。

7.6.3　Apache Mesos

Apache Mesos 是一个通用集群管理器，既可以运行分析型工作负载又可以运行长期运行的服务（比如网页服务或者键值对存储）。要在 Mesos 上使用 Spark，需要把一个 mesos:// 的 URI 传给 `spark-submit`：

```
spark-submit --master mesos://masternode:5050 yourapp
```

在运行多个主节点时，你可以使用 ZooKeeper 来为 Mesos 集群选出一个主节点。在这种情况下，应该使用 mesos://zk:// 的 URI 来指向一个 ZooKeeper 节点列表。比如，你有三个 ZooKeeper 节点

（node1、node2 和 node3），并且 ZooKeeper 分别运行在三台机器的 2181 端口上时，你应该使用如下 URI：

```
mesos://zk://node1:2181/mesos,node2:2181/mesos,node3:2181/mesos
```

1. Mesos调度模式

和别的集群管理器不同，Mesos 提供了两种模式来在一个集群内的执行器进程间共享资源。在“细粒度”模式（默认）中，执行器进程占用的 CPU 核心数会在它们执行任务时动态变化，因此一台运行了多个执行器进程的机器可以动态共享 CPU 资源。而在“粗粒度”模式中，Spark 提前为每个执行器进程分配固定数量的 CPU 数目，并且在应用结束前绝不释放这些资源，哪怕执行器进程当前没有运行任务。你可以通过向 spark-submit 传递 --conf spark.mesos.coarse=true 来打开粗粒度模式。

当多用户共享的集群运行 shell 这样的交互式的工作负载时，由于应用会在它们不工作时降低它们所占用的核心数，以允许别的用户程序使用集群，所以这种情况下细粒度模式显得非常合适。然而，在细粒度模式下调度任务会带来更多的延迟（这样的话，一些像 Spark Streaming 这样需要低延迟的应用就会表现很差），应用需要在用户输入新的命令时，为重新分配 CPU 核心等待一段时间。不过值得一提的是，你可以在一个 Mesos 集群中使用混合的调度模式（比如将一部分 Spark 应用的 spark.mesos.coarse 设置为 true，而另一部分不这么设置）。

2. 客户端和集群模式

就 Spark 1.2 而言，在 Mesos 上 Spark 只支持以客户端的部署模式运行应用。也就是说，驱动器程序必须运行在提交应用的那台机器上。如果你还是希望在 Mesos 集群中运行你的驱动器节点，那么 Aurora（http://aurora.apache.org/）或 Chronos（http://airbnb.io/chronos）这样的框架可以让你将任意脚本提交到 Mesos 上执行，并监控它们的运行。你可以使用这类框架中的一种来启动应用的驱动器程序。

3. 配置资源用量

你可以通过 spark-submit 的两个参数 --executor-memory 和 --total-executor-cores 来控制运行在 Mesos 上的资源用量，

前者用来设置每个执行器进程的内存，后者则用来设置应用占用的核心数（所有执行器节点占用的总数）的最大值。默认情况下，Spark 会使用尽可能多的核心启动各个执行器节点，来将应用合并到尽量少的执行器实例中，并为应用分配所需要的核心数。如果你不设置 `--total-executor-cores` 参数，Mesos 会尝试使用集群中所有可用的核心。

7.6.4 Amazon EC2

Spark 自带一个可以在 Amazon EC2 上启动集群的脚本。这个脚本会启动一些节点，并且在它们上面安装独立集群管理器。这样一旦集群启动起来，你就可以根据前面讲到的独立模式使用方法来使用这个集群。除此以外，EC2 脚本还会安装好其他相关的服务，比如 HDFS、Tachyon 还有用来监控集群的 Ganglia。

Spark 的 EC2 脚本叫作 `spark-ec2`，位于 Spark 安装目录下的 ec2 文件夹中。它需要 Python 2.6 或更高版本的运行环境。你可以在下载 Spark 后直接运行 EC2 脚本而无需预先编译 Spark。

EC2 脚本可以管理多个已命名的集群（cluster），并且使用 EC2 安全组来区分它们。对于每个集群，脚本都会为主节点创建一个叫作 `clustername-master` 的安全组，而为工作节点创建一个叫作 `clustername-slaves` 的安全组。

1. 启动集群

要启动一个集群，你应该先创建一个 Amazon 网络服务（AWS）账号，并且获取访问键 ID 和访问键密码，然后把它们设在环境变量中：

```
export AWS_ACCESS_KEY_ID="..."
export AWS_SECRET_ACCESS_KEY="..."
```

然后再创建出 EC2 的 SSH 密钥对，然后下载私钥文件（通常叫作 keypair.pem），这样你就可以 SSH 到你的机器上。

接下来，运行 `spark-ec2` 脚本的 `launch` 命令，提供你的密钥对的名字、私钥文件和集群的名字。默认情况下，这条命令会使用 `m1.xlarge` 类型的 EC2 实例，启动一个有一个主节点和一个工作节点的集群：

```
cd /path/to/spark/ec2
./spark-ec2 -k mykeypair -i mykeypair.pem launch mycluste
```

你也可以使用 `spark-ec2` 的参数选项配置实例的类型、工作节点个数、EC2 地区，以及其他一些要素。例如：

```
# 启动包含5个m3.xlarge类型的工作节点的集群
./spark-ec2 -k mykeypair -i mykeypair.pem -s 5 -t m3.xlarge launch myclust
```

要获得参数选项的完整列表，运行 `spark-ec2 --help`。表 7-3 列出了一些常用的选项。

表7-3：**spark-ec2**的常见选项

选项	含义
-k KEYPAIR	要使用的 keypair 的名字
-i IDENTITY_FILE	私钥文件（以 .pem 结尾）
-s NUM_SLAVES	工作节点数量
-t INSTANCE_TYPE	使用的实例类型
-r REGION	使用 Amazon 实例所在的区域（例如 us-west-1）
-z ZONE	使用的地带（例如 us-west-1b）
--spot-price=PRICE	在给定的出价使用 spot 实例（单位为美元）

从启动脚本开始，通常需要五分钟左右来完成启动机器、登录到机器上并配置 Spark 的全部过程。

2. 登录集群

你可以使用存有私钥的 .pem 文件通过 SSH 登录到集群的主节点上。为了方便，`spark-ec2` 提供了登录命令：

```
./spark-ec2 -k mykeypair -i mykeypair.pem login mycluster
```

还有，你可以通过运行下面的命令获得主节点的主机名：

```
./spark-ec2 get-master mycluster
```

然后自行使用 `ssh -i keypair.pem root@masternode` 命令 SSH 到主节点上。

进入集群以后，你就可以使用 /root/spark 中的 Spark 环境来运行程序了。这是一个独立模式的集群环境，主节点 URL 是 spark://masternode:7077。当你使用 spark-submit 启动应用时，Spark 会自动配置为将应用提交到这个独立集群上。你可以从 http://masternode:8080 访问集群的网页用户界面。

注意，只有从集群中的机器上启动的程序可以使用 spark-submit 把作业提交上去；出于安全目的，防火墙规则会阻止外部主机提交作业。要在集群上运行一个预先打包的应用，需要先把程序包通过 SCP 复制到集群上：

```
scp -i mykeypair.pem app.jar root@masternode:~
```

3. 销毁集群

要销毁 spark-ec2 所启动的集群，运行：

```
./spark-ec2 destroy mycluster
```

这条命令会终止集群中的所有的实例（包括 mycluster-master 和 mycluster-slaves 两个安全组中的所有实例）。

4. 暂停和重启集群

除了将集群彻底销毁，spark-ec2 还可以让你中止运行集群的 Amazon 实例，并且可以让你稍后重启这些实例。停止这些实例会将它们关机，并丢失“临时”盘上的所有数据。“临时”盘上还配有一个给 spark-ec2 使用的 HDFS 环境（参见下一节）。不过，中止的实例会 保留 root 目录下的所有数据（例如你上传到那里的所有文件），这样你就可以快速恢复自己的工作。

要中止一个集群，运行：

```
./spark-ec2 stop mycluster
```

然后，过一会儿，再次启动集群：

```
./spark-ec2 -k mykeypair -i mykeypair.pem start mycluster
```

 Spark EC2 的脚本并没有提供调整集群大小的命令，但你可以通过增减 `mycluster-slaves` 安全组中的机器来实现对集群大小的控制。要增加机器，首先应当中止集群，然后使用 AWS 管理控制台，右击一台工作节点并选择"Launch more like this（启动更多像这个实例一样的实例）"。这样我们就可以在同一个安全组中创建出更多实例。然后使用 `spark-ec2 start` 启动集群。要移除机器，只需在 AWS 控制台上终止这一实例即可（不过要小心，这也会破坏集群中 HDFS 上的数据）。

5. 集群存储

Spark EC2 集群已经配置好了两套 Hadoop 文件系统以供存储临时数据。这样我们可以很方便地将数据集存储在访问速度比 Amazon S3 更快的媒介中。这两套文件系统详述如下。

- "临时"HDFS，使用节点上的临时盘。大多数类型的 Amazon 实例都在"临时"盘上带有大容量的本地空间，这些空间会在实例关机时消失。"临时"HDFS 使用这种空间，可以提供相当大的临时存储空间。不过它会在你关闭并重启 EC2 集群时丢失所有数据。这种文件系统安装在节点的 /root/ephemeral-hdfs 目录中，你可以使用 `bin/hdfs` 命令来访问并列出文件。你也可以访问这种 HDFS 的网页用户界面，其 URL 地址位于 http://masternode:50070。

- "永久"HDFS，使用节点的 root 分卷。这种 HDFS 在集群重启时仍然保留数据，不过一般空间较小，访问起来也比临时的那种慢。这种 HDFS 适合存放你不想多次下载的中等大小的数据集中。它安装于 /root/persistent-hdfs 目录，网页用户界面地址是 http://masternode:60070。

除了这些以外，你最有可能访问的就是 Amazon S3 中的数据了。你可以在 Spark 中使用 s3n:// 的 URI 结构来访问其中的数据，具体细节请参考 5.3.2 节。

7.7　选择合适的集群管理器

Spark 所支持的各种集群管理器为我们提供了部署应用的多种选择。如果你需要从零开始部署，正在权衡各种集群管理器，我们推荐如下

一些准则。

- 如果是从零开始，可以先选择独立集群管理器。独立模式安装起来最简单，而且如果你只是使用 Spark 的话，独立集群管理器提供与其他集群管理器完全一样的全部功能。

- 如果你要在使用 Spark 的同时使用其他应用，或者是要用到更丰富的资源调度功能（例如队列），那么 YARN 和 Mesos 都能满足你的需求。而在这两者中，对于大多数 Hadoop 发行版来说，一般 YARN 已经预装好了。

- Mesos 相对于 YARN 和独立模式的一大优点在于其细粒度共享的选项，该选项可以将类似 Spark shell 这样的交互式应用中的不同命令分配到不同的 CPU 上。因此这对于多用户同时运行交互式 shell 的用例更有用处。

- 在任何时候，最好把 Spark 运行在运行 HDFS 的节点上，这样能快速访问存储。你可以自行在同样的节点上安装 Mesos 或独立集群管理器。如果使用 YARN 的话，大多数发行版已经把 YARN 和 HDFS 安装在了一起。

最后，请牢记集群管理器仍处于快速发展中。在这本书面世之际，Spark 当前支持的这些集群管理器可能已经有了一些针对 Spark 的新特性，而 Spark 也有可能已经增加了对新的集群管理器的支持。本章中描述的提交应用的方法不会改变，不过你依然应当查阅所使用 Spark 版本的官方文档（http://spark.apache.org/docs/latest/）来了解最新的选项。

7.8　总结

本章描述了 Spark 应用的运行时架构，它是由一个驱动器节点和一系列分布式执行器节点组成的。之后本章讲了如何构建、打包 Spark 应用并向集群提交执行。最后，我们总结了常见的 Spark 部署环境，包括 Spark 自带的集群管理器，还有在 YARN 或 Mesos 等第三方集群管理器上运行 Spark，以及在 Amazon EC2 云服务上运行。下一章会深入介绍更多操作细节，主要集中于调优和调试生产环境中的 Spark 应用。

第 8 章　Spark 调优与调试

本章介绍如何配置 Spark 应用，并粗略介绍如何调优和调试生产环境中的 Spark 工作负载。尽管 Spark 的默认设置在很多情况下无需修改就能直接使用，但有时我们也确实需要修改某些选项。本章会总结 Spark 的配置机制，着重讨论用户可能需要稍作调整的一些选项。Spark 的配置对于应用的性能调优是很有意义的。本章的第二部分则涵盖了理解 Spark 应用性能表现的必备基础知识，以及设置相关配置项、编写高性能应用的设计模式等。我们也会讨论 Spark 的用户界面、执行的组成部分、日志机制等相关内容。这些信息在进行性能调优和问题追踪时都是非常有用的。

8.1　使用SparkConf配置Spark

对 Spark 进行性能调优，通常就是修改 Spark 应用的运行时配置选项。Spark 中最主要的配置机制是通过 `SparkConf` 类对 Spark 进行配置。当创建出一个 SparkContext 时，就需要创建出一个 `SparkConf` 的实例，如例 8-1 至例 8-3 所示。

例 **8-1**：在 Python 中使用 SparkConf 创建一个应用

```
# 创建一个conf对象
conf = new SparkConf()
conf.set("spark.app.name", "My Spark App")
conf.set("spark.master", "local[4]")
conf.set("spark.ui.port", "36000") # 重载默认端口配置

# 使用这个配置对象创建一个SparkContext
sc = SparkContext(conf)
```

例 **8-2**：在 Scala 中使用 SparkConf 创建一个应用

```
// 创建一个conf对象
val conf = new SparkConf()
conf.set("spark.app.name", "My Spark App")
conf.set("spark.master", "local[4]")
conf.set("spark.ui.port", "36000") // 重载默认端口配置

// 使用这个配置对象创建一个SparkContext
val sc = new SparkContext(conf)
```

例 **8-3**：在 Java 中使用 SparkConf 创建一个应用

```
// 创建一个conf对象
SparkConf conf = new SparkConf();
conf.set("spark.app.name", "My Spark App");
conf.set("spark.master", "local[4]");
conf.set("spark.ui.port", "36000"); // 重载默认端口配置

// 使用这个配置对象创建一个SparkContext
JavaSparkContext sc = JavaSparkContext(conf);
```

`SparkConf` 类其实很简单：`SparkConf` 实例包含用户要重载的配置选项的键值对。Spark 中的每个配置选项都是基于字符串形式的键值对。要使用创建出来的 `SparkConf` 对象，可以调用 `set()` 方法来添加配置项的设置，然后把这个对象传给 SparkContext 的构造方法。除了 `set()` 之外，`SparkConf` 类也包含了一小部分工具方法，可以很方便地设置部分常用参数。例如，在前面的三个例子中，你也可以调用 `setAppName()` 和 `setMaster()` 来分别设置 `spark.app.name` 和 `spark.master` 的配置值。

在这几个例子中，`SparkConf` 中的值都是在应用代码中设置的。在很多情况下，动态地为给定应用设置配置选项会方便得多。Spark 允许通过 `spark-submit` 工具动态设置配置项。当应用被 `spark-submit` 脚本启动时，脚本会把这些配置项设置到运行环境中。当一个新的 `SparkConf` 被创建出来时，这些环境变量会被检测出来并且自动配好。这样，在使用 `spark-submit` 时，用户应用中只要创建一个“空”的 `SparkConf`，并直接传给 SparkContext 的构造方法就行了。

`spark-submit` 工具为常用的 Spark 配置项参数提供了专用的标记，还有一个通用标记 `--conf` 来接收任意 Spark 配置项的值，如例 8-4 所示。

例 **8-4**：在运行时使用标记设置配置项的值

```
$ bin/spark-submit \
  --class com.example.MyApp \
  --master local[4] \
  --name "My Spark App" \
  --conf spark.ui.port=36000 \
  myApp.jar
```

spark-submit 也支持从文件中读取配置项的值。这对于设置一些与环境相关的配置项比较有用，方便不同用户共享这些配置（比如默认的 Spark 主节点）。默认情况下，spark-submit 脚本会在 Spark 安装目录中找到 conf/spark-defaults.conf 文件，尝试读取该文件中以空格隔开的键值对数据。你也可以通过 spark-submit 的 --properties-File 标记，自定义该文件的路径，如例 8-5 所示。

例 8-5：运行时使用默认文件设置配置项的值

```
$ bin/spark-submit \
  --class com.example.MyApp \
  --properties-file my-config.conf \
  myApp.jar

## Contents of my-config.conf ##
spark.master     local[4]
spark.app.name   "My Spark App"
spark.ui.port    36000
```

一旦传给了 SparkContext 的构造方法，应用所绑定的 SparkConf 就不可变了。这意味着所有的配置项都必须在 SparkContext 实例化出来之前定下来。

有时，同一个配置项可能在多个地方被设置了。例如，某用户可能在程序代码中直接调用了 setAppName() 方法，同时也通过 spark-submit 的 --name 标记设置了这个值。针对这种情况，Spark 有特定的优先级顺序来选择实际配置。优先级最高的是在用户代码中显式调用 set() 方法设置的选项。其次是通过 spark-submit 传递的参数，再次是写在配置文件中的值，最后是系统的默认值。如果你想获得应用中实际生效的配置，可以在应用的网页用户界面中查看，本章稍后会作进一步讨论。

表 7-2 列出了一些常用的配置项。表 8-1 则列出了其他几个比较值得关注的配置项。如果想要得到完整的配置项列表，请参考 Spark 文档（http://spark.apache.org/docs/latest/configuration.html）。

表8-1：常用的Spark配置项的值

选项	默认值	描述
spark.executor.memory	512m	为每个执行器进程分配的内存，格式与 JVM 内存字符串格式一样（例如 512m，2g）。关

		于本配置项的更多细节，请参阅 8.4.4 节
spark.executor.cores / spark.cores.max	1 / (无)	限制应用使用的核心个数的配置项。在 YARN 模式下，spark.executor.cores 会为每个任务分配指定数目的核心。在独立模式和Mesos 模式下，spark.core.max 设置了所有执行器进程使用的核心总数的上限。参阅 8.4.4 节了解更多细节
spark.speculation		设为 true 时开启任务预测执行机制。当出现比较慢的任务时，这种机制会在另外的节点上也尝试执行该任务的一个副本。打开此选项会帮助减少大规模集群中个别较慢的任务带来的影响
spark.storage.blockManagerTimeoutIntervalMs		内部用来通过超时机制追踪执行器进程是否存活的阈值。对于会引发长时间垃圾回收（GC）暂停的作业，需要把这个值调到 100 秒（对应值为 100000）以上来防止失败。在 Spark 将来的版本中，这个配置项可能会被一个统一的超时设置所取代，所以请注意检索最新文档
spark.executor.extraJavaOptions spark.executor.extraClassPath spark.executor.extraLibraryPath	(空)	这三个参数用来自定义如何启动执行器进程的 JVM，分别用来添加额外的 Java 参数、classpath 以及程序库路径。使用字符串来设置这些参数（例如 spark.executor.extraJavaOptions="- XX:+PrintGCDetails-XX:+PrintGCTimeStamps"）。请注意，虽然这些参数可以让你自行添加执行器程序的 classpath，我们还是推荐使用 spark-submit 的 --jars 标记来添加依赖而不是使用这几个选项
spark.serializer	org.apache.spark.serializer.JavaSerializer	指定用来进行序列化的类库，包括通过网络传输数据或缓存数据时的序列化。默认的 Java 序列化对于任何可以被序列化的 Java 对象都适用，但是速度很慢。我们推荐在追求速度时使用 org.apache.spark.serializer.KryoSerializer 并且对 Kryo 进行适当的调优。该项可以配置为任何 org.apache.spark.Serializer 的子类
spark.[X].port	(任意值)	用来设置运行 Spark 应用时用到的各个端口。这些参数对于运行在可靠网络上的集群是很有用的。有效的 X 包括 driver、fileserver、broadcast、replClassServer、blockManager，以及 executor
spark.eventLog.enabled		设为 true 时，开启事件日志机制，这样已完成的 Spark 作业就可以通过历史服务器（history server）查看。关于历史服务器的更多信息，请参考官方文档
spark.eventLog.dir		指开启事件日志机制时，事件日志文件的存储位置。这个值指向的路径需要设置到一个全局可见的文件系统中，比如 HDFS

几乎所有的 Spark 配置都发生在 SparkConf 的创建过程中，但有一个重要的选项是个例外。你需要在 conf/spark-env.sh 中将环境变量 SPARK_LOCAL_DIRS 设置为用逗号隔开的存储位置列表，来指定 Spark 用来混洗数据的本地存储路径。这需要在独立模式和 Mesos 模式下设置。8.4.4 节中会更详细地介绍 SPARK_LOCAL_DIRS 的相关知识点。这个配置项之所以和其他的 Spark 配置项不一样，是因为它的值在不同的物理主机上可能会有区别。

8.2　Spark执行的组成部分：作业、任务和步骤

要对 Spark 进行调优和调试，首先要进一步了解系统的内部设计。在前面的章节中，你已经看到了 RDD 的逻辑表示以及 RDD 的分区。在执行时，Spark 会把多个操作合并为一组任务，把 RDD 的逻辑表示翻译为物理执行计划。理解 Spark 执行的方方面面不在本书要讨论的范围内，不过理解一些执行过程中涉及的流程以及相关的术语对于调

优和调试是非常有帮助的。

下面通过一个示例应用来展示 Spark 执行的各个阶段，以了解用户代码如何被编译为下层的执行计划。我们要使用的是一个使用 Spark shell 实现的简单的日志分析应用。输入数据是一个由不同严重等级的日志消息和一些分散的空行组成的文本文件，如例 8-6 所示。

例 **8-6**：用作示例的源文件 input.txt

```
## input.txt ##
INFO This is a message with content
INFO This is some other content
（空行）
INFO Here are more messages
WARN This is a warning
（空行）
ERROR Something bad happened
WARN More details on the bad thing
INFO back to normal messages
```

我们希望在 Spark shell 中打开该文件，计算其中各级别的日志消息的条数。首先我们来创建几个可以帮助我们回答该问题的 RDD，如例 8-7 所示。

例 **8-7**：在 Scala 版本的 Spark shell 中处理文本数据

```
// 读取输入文件
scala> val input = sc.textFile("input.txt")
// 切分为单词并且删掉空行
scala> val tokenized = input.
     |   map(line => line.split(" ")).
     | filter(words => words.size > 0)
// 提取出每行的第一个单词（日志等级）并进行计数
scala> val counts = tokenized.
     |   map(words = > (words(0), 1)).
     | reduceByKey{ (a,b) => a + b }
```

这一系列命令生成了一个叫作 `counts` 的 RDD，其中包含各级别日志对应的条目数。在 shell 中执行完这些命令之后，程序没有执行任何行动操作。相反，程序定义了一个 RDD 对象的有向无环图（DAG），我们可以在稍后行动操作被触发时用它来进行计算。每个 RDD 维护了其指向一个或多个父节点的引用，以及表示其与父节点之间关系的信息。比如，当你在 RDD 上调用 `val b = a.map()`

时，b 这个 RDD 就存下了对其父节点 a 的一个引用。这些引用使得 RDD 可以追踪到其所有的祖先节点。

Spark 提供了 toDebugString() 方法来查看 RDD 的谱系。在例 8-8 中，我们会看到在前面的例子中创建出的一些 RDD。

例 **8-8**：在 Scala 中使用 toDebugString() 查看 RDD

```
scala> input.toDebugString
res85: String =
(2) input.text MappedRDD[292] at textFile at <console>:13
  | input.text HadoopRDD[291] at textFile at <console>:13

scala> counts.toDebugString
res84: String =
(2) ShuffledRDD[296] at reduceByKey at <console>:17
 +-(2) MappedRDD[295] at map at <console>:17
    |  FilteredRDD[294] at filter at <console>:15
    |  MappedRDD[293] at map at <console>:15
    |  input.text MappedRDD[292] at textFile at <console>:13
    |  input.text HadoopRDD[291] at textFile at <console>:13
```

第一条命令输出了 RDDinput 的相关信息。通过调用 sc.textFile() 创建出了这个 RDD。从输出的谱系中，可以看到 sc.textFile() 方法所创建出的 RDD 类型，找到关于 textFile() 幕后细节的一些蛛丝马迹。事实上，该方法创建出了一个 HadoopRDD 对象，然后对该 RDD 执行映射操作，最终得到了返回的 RDD。counts 的谱系图则更加复杂一些，有多个祖先 RDD，这是由于我们对 input 进行了其他操作，包括额外的映射、筛选以及归约。counts 谱系也以图的方式展示在图 8-1 左侧。

在调用行动操作之前，RDD 都只是存储着可以让我们计算出具体数据的描述信息。要触发实际计算，需要对 counts 调用一个行动操作，比如使用 collect() 将数据收集到驱动器程序中，如例 8-9 所示。

例 **8-9**：收集 RDD

```
scala> counts.collect()
res86: Array[(String, Int)] = Array((ERROR,1), (INFO,4), (WARN,2))
```

Spark 调度器会创建出用于计算行动操作的 RDD 物理执行计划。我们在此处调用 RDD 的 `collect()` 方法，于是 RDD 的每个分区都会被物化出来并发送到驱动器程序中。Spark 调度器从最终被调用行动操作的 RDD（在本例中是 `counts`）出发，向上回溯所有必须计算的 RDD。调度器会访问 RDD 的父节点、父节点的父节点，以此类推，递归向上生成计算所有必要的祖先 RDD 的物理计划。我们以最简单的情况为例，调度器为有向图中的每个 RDD 输出计算步骤，步骤中包括 RDD 上需要应用于每个分区的任务。然后以相反的顺序执行这些步骤，计算得出最终所求的 RDD。

下面来看看更复杂的情况，此时 RDD 图与执行步骤的对应关系并不一定是一一对应的，比如，当调度器进行流水线执行（pipelining），或把多个 RDD 合并到一个步骤中时。当 RDD 不需要混洗数据就可以从父节点计算出来时，调度器就会自动进行流水线执行。例 8-8 中输出的谱系图使用不同缩进等级来展示 RDD 是否会在物理步骤中进行流水线执行。在物理执行时，执行计划输出的缩进等级与其父节点相同的 RDD 会与其父节点在同一个步骤中进行流水线执行。例如，当计算 `counts` 时，尽管有很多级父 RDD，但从缩进来看总共只有两级。这表明物理执行只需要两个步骤。由于执行序列中有几个连续的筛选和映射操作，这个例子中才出现了流水线执行。图 8-1 的右半部分展示了计算 `counts` 这个 RDD 时的两个执行步骤。

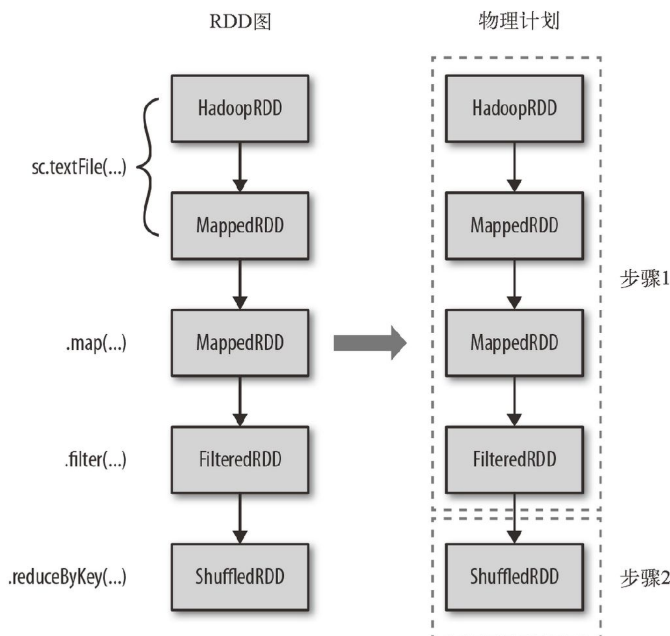


图 **8-1**：将 **RDD** 转化操作串联成物理执行步骤

如果访问应用的网页用户界面，你会发现 `collect()` 操作引发了两个步骤。如果你在自己的机器上运行这段示例代码，用户界面可以在（http://localhost:4040）上找到。本章后半部分会更详细地讨论用户界面的细节，现在你只需要简单看看程序运行期间执行了哪些步骤。

除了流水线执行的优化，当一个 RDD 已经缓存在集群内存或磁盘上时，Spark 的内部调度器也会自动截短 RDD 谱系图。在这种情况下，Spark 会“短路”求值，直接基于缓存下来的 RDD 进行计算。还有一种截短 RDD 谱系图的情况发生在当 RDD 已经在之前的数据混洗中作为副产品物化出来时，哪怕该 RDD 并没有被显式调用 `persist()` 方法。这种内部优化是基于 Spark 数据混洗操作的输出均被写入磁盘的特性，同时也充分利用了 RDD 图的某些部分会被多次计算的事实。

下面来看看物理执行中缓存的作用。我们把 `counts`RDD 缓存下来，观察之后的行动操作是怎样被截短的（例 8-10）。如果你再次访问用户界面，你会看到缓存减少了后来执行计算时所需要的步骤。多次调用 `collect()` 只会产生一个步骤来完成这个行动操作。

例 **8-10**：计算一个已经缓存过的 RDD

```
// 缓存RDD
scala> counts.cache()
// 第一次求值运行仍然需要两个步骤
scala> counts.collect()
res87: Array[(String, Int)] = Array((ERROR,1), (INFO,4), (WARN,2), (##,1),
((empty,2))
// 该次执行只有一个步骤
scala> counts.collect()
res88: Array[(String, Int)] = Array((ERROR,1), (INFO,4), (WARN,2), (##,1),
((empty,2))
```

特定的行动操作所生成的步骤的集合被称为一个作业。我们通过类似 `count()` 之类的方法触发行动操作，创建出由一个或多个步骤组成的作业。

一旦步骤图确定下来，任务就会被创建出来并发给内部的调度器。该调度器在不同的部署模式下会有所不同。物理计划中的步骤会依赖于其他步骤，如 RDD 谱系图所显示的那样。因此，这些步骤会以特定的顺序执行。例如，一个输出混洗后的数据的步骤一定会依赖于进行数据混洗的那个步骤。

一个物理步骤会启动很多任务，每个任务都是在不同的数据分区上做同样的事情。任务内部的流程是一样的，如下所述。

(1) 从数据存储（如果该 RDD 是一个输入 RDD）或已有 RDD（如果该步骤是基于已经缓存的数据）或数据混洗的输出中获取输入数据。

(2) 执行必要的操作来计算出这些操作所代表的 RDD。例如，对输入数据执行 `filter()` 和 `map()` 函数，或者进行分组或归约操作。

(3) 把输出写到一个数据混洗文件中，写入外部存储，或者是发回驱动器程序（如果最终 RDD 调用的是类似 `count()` 这样的行动操作）。

Spark 的大部分日志信息和工具都是以步骤、任务或数据混洗为单位的。理解用户代码如何编译为物理执行的内容是一个高深的话题，但对调优和调试应用有非常大的帮助。

归纳一下，Spark 执行时有下面所列的这些流程。

- 用户代码定义**RDD**的有向无环图

RDD 上的操作会创建出新的 RDD，并引用它们的父节点，这样就创建出了一个图。

- 行动操作把有向无环图强制转译为执行计划

  当你调用 RDD 的一个行动操作时，这个 RDD 就必须被计算出来。这也要求计算出该 RDD 的父节点。Spark 调度器提交一个作业来计算所有必要的 RDD。这个作业会包含一个或多个步骤，每个步骤其实也就是一波并行执行的计算任务。一个步骤对应有向无环图中的一个或多个 RDD，一个步骤对应多个 RDD 是因为发生了流水线执行。

- 任务于集群中调度并执行

  步骤是按顺序处理的，任务则独立地启动来计算出 RDD 的一部分。一旦作业的最后一个步骤结束，一个行动操作也就执行完毕了。

在一个给定的 Spark 应用中，由于需要创建一系列新的 RDD，因此上述阶段会连续发生很多次。

8.3 查找信息

Spark 在应用执行时记录详细的进度信息和性能指标。这些内容可以在两个地方找到： Spark 的网页用户界面以及驱动器进程和执行器进程生成的日志文件中。

8.3.1 Spark网页用户界面

Spark 内建的网页用户界面是了解 Spark 应用的行为和性能表现的第一站。默认情况下，它在驱动器程序所在机器的 4040 端口上。不过对于 YARN 集群模式来说，应用的驱动器程序会运行在集群内部，你应该通过 YARN 的资源管理器来访问用户界面。YARN 的资源管理器会把请求直接转发给驱动器程序。

Spark 用户界面包括几个不同的页面，具体的格式在不同的 Spark 版本间也存在区别。以 bSpark 1.2 为例，用户界面主要由四个不同的页面组成，下面一一进行讨论。

1. 作业页面：步骤与任务的进度和指标，以及更多内容

如图 8-2 所示，作业页面包含正在进行的或刚完成不久的 Spark 作业的详细执行情况。其 中一个很重要的信息是正在运行的作业、步骤以及任务的进度情况。针对每个步骤，这个 页面提供了一些帮助理解物理执行过程的指标。

作业页面是在 Spark 1.2 中才引入的，如果你使用更早版本的话，就看不到了。

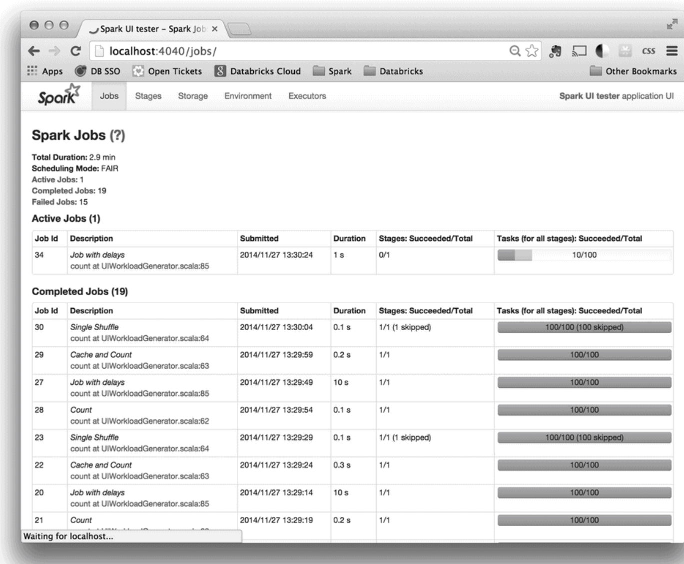


图 **8-2**：**Spark** 应用用户界面的作业索引页面

本页面经常用来评估一个作业的性能表现。我们可以着眼于组成作业的所有步骤，看看是不是有一些步骤特别慢，或是在多次运行同一个作业时响应时间差距很大。如果你遇到了格外慢的步骤，你可以点击进去来更好地理解该步骤对应的是哪段用户代码。

确定了需要着重关注的步骤之后，你可以再借助图 8-3 所示的步骤页面来定位性能问题。在 Spark 这样的并行数据系统中，数据倾斜是导致性能问题的常见原因之一。当看到少量的任务相对于其他任务需要花费大量时间的时候，一般就是发生了数据倾斜。步骤页面可以帮助我们发现数据倾斜，我们只需要查看所有任务各项指标的分布情况就可以了。我们可以从任务的运行时间开始看。是不是有一部分任务花的时间比别的任务多得多？如果是的话，你可以深挖下去，看到底是

什么导致了这些任务运行缓慢。是不是有一小部分任务读取或者输出了比别的任务多得多的数据？是不是运行在某些特定节点上的任务特别慢？这些都可以看作是在对作业进行调试时很有帮助的出发点。

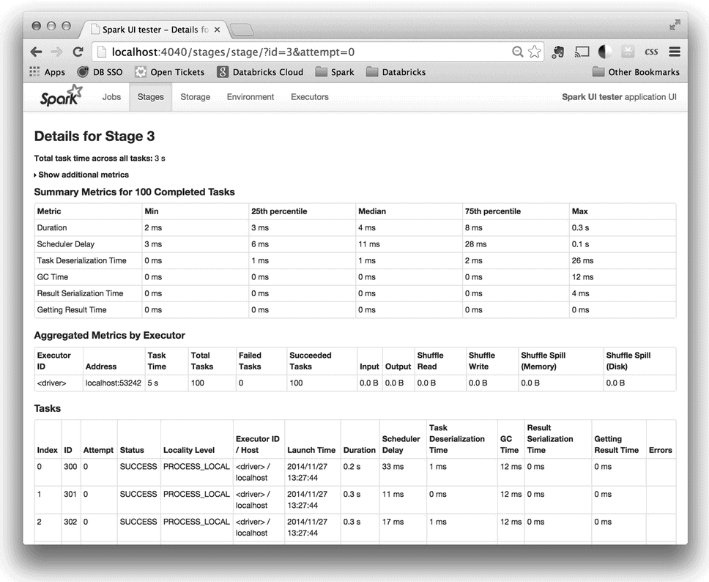


图 **8-3**：**Spark** 应用用户界面中的步骤详情页面

除了发现数据倾斜，本页面还可以用来查看任务在其生命周期的各个阶段（读取、计算、输出）分别花费了多少时间。如果任务花了很少的时间读取或输出数据，但是总时间却很长，这就可能表明用户代码本身的执行比较花时间（用户代码优化的例子请参见 6.4 节）。有些任务可能把时间基本都花在从外部存储系统中读取数据这部分了，这样为 Spark 作额外的优化对于提高性能就没有多大用处了，毕竟这些任务的瓶颈在于输入数据的读取。

2. 存储页面：已缓存的**RDD**的信息

存储页面包含了缓存下来的 RDD 的信息。当有人在一个 RDD 上调用了 `persist()` 方法，并且在某个作业中计算了该 RDD 时，这个 RDD 就会被缓存下来。有时，如果我们缓存了许多 RDD，比较老的 RDD 就会从内存中移出来，把空间留给新缓存的 RDD。这个页面可以告诉我们到底各个 RDD 的哪些部分被缓存了，以及在各种不同的存储媒介（磁盘、内存等）中所缓存的数据量。浏览这个页面并理解

一些重要的数据集是否被缓存在了内存中，对我们是很有意义的。

3. 执行器页面：应用中的执行器进程列表

本页面列出了应用中申请到的执行器实例，以及各执行器进程在数据处理和存储方面的一些指标。本页面的用处之一在于确认应用可以使用你所预期使用的全部资源量。调试问题时也最好先浏览这个页面，因为错误的配置可能会导致启动的执行器进程数量少于我们所预期的，显然也就会影响实际性能。这个页面对于查找行为异常的执行器节点也很有帮助，比如某个执行器节点可能有很高的任务失败率。失败率很高的执行器节点可能表明这个执行器进程所在的物理主机的配置有问题或者出了故障。只要把这台主机从集群中移除，就可以提高性能表现。

执行器页面的另一个功能是使用线程转存（Thread Dump）按钮收集执行器进程的栈跟踪信息（该功能在 Spark 1.2 中引入）。可视化呈现执行器进程的线程调用栈可以精确地即时显示出当前执行的代码。在短时间内使用该功能对一个执行器进程进行多次采样，你就可以发现"热点"，也就是用户代码中消耗代价比较大的代码段。这种信息分析通常可以检测出低效的用户代码。

4. 环境页面：用来调试**Spark**配置项

本页面枚举了你的 Spark 应用所运行的环境中实际生效的配置项集合。这里显示的配置项代表应用实际的配置情况。当你检查哪些配置标记生效时，这个页面很有用，尤其是当你同时使用了多种配置机制时。这个页面也会列出你添加到应用路径中的所有 JAR 包和文件，在追踪类似依赖缺失的问题时可以用到。

8.3.2 驱动器进程和执行器进程的日志

在某些情况下，用户需要深入研读驱动器进程和执行器进程所生成的日志来获取更多信息。日志会更详细地记录各种异常事件，例如内部的警告以及用户代码输出的详细异常信息。这些数据对于寻找错误原因很有用。

Spark 日志文件的具体位置取决于以下部署模式。

- 在 Spark 独立模式下，所有日志会在独立模式主节点的网页用户界面中直接显示。这些日志默认存储于各个工作节点的 Spark 目录下的 work/ 目录中。

- 在 Mesos 模式下，日志存储在 Mesos 从节点的 work/ 目录中，可以通过 Mesos 主节点用户界面访问。

- 在 YARN 模式下，最简单的收集日志的方法是使用 YARN 的日志收集工具（运行 `yarn logs -applicationId <app ID>`）来生成一个包含应用日志的报告。这种方法只有在应用已经完全完成之后才能使用，因为 YARN 必须先把这些日志聚合到一起。要查看当前运行在 YARN 上的应用的日志，你可以从资源管理器的用户界面点击进入节点（Nodes）页面，然后浏览特定的节点，再从那里找到特定的容器。YARN 会提供对应容器中 Spark 输出的内容以及相关日志。将来的 Spark 版本有可能会提供直接指向相应日志的链接，使得这一过程显得不再那么迂回。

在默认情况下，Spark 输出的日志包含的信息量比较合适。我们也可以自定义日志行为，改变日志的默认等级或者默认存放位置。Spark 的日志系统是基于广泛使用的 Java 日志库 log4j 实现的，使用 log4j 的配置方式进行配置。log4j 配置的示例文件已经打包在 Spark 中，具体位置是 conf/log4j.properties.template。要想自定义 Spark 的日志，首先需要把这个示例文件复制为 log4j.properties，然后就可以修改日志行为了，比如修改根日志等级（即日志输出的级别门槛），默认值为 `INFO`。如果想要更少的日志输出，可以把该值设为 `WARN` 或者 `ERROR`。当设置了满意的日志等级或格式之后，你可以通过 `spark-submit` 的 `--Files` 标记添加 log4j.properties 文件。如果你在设置日志级别时遇到了困难，请首先确保你没有在应用中引入任何自身包含 log4j.properties 文件的 JAR 包。Log4j 会扫描整个 classpath，以其找到的第一个配置文件为准，因此如果在别处先找到该文件，它就会忽略你自定义的文件。

8.4　关键性能考量

读到这里，你应该已经掌握了一些 Spark 的内部工作原理，如何跟踪运行中的 Spark 应用的进度，以及在哪里查看指标和日志信息。本节将进入下一步，讨论运行 Spark 应用时可能会遇到的性能方面的常见问题，以及关于如何调优应用以获取最佳性能的一些小提示。前三小节会介绍如何在代码层面进行改动来提高性能，最后一小节则会讨论如何调优集群设定以及 Spark 的运行环境。

8.4.1　并行度

RDD 的逻辑表示其实是一个对象集合。我们在本书中已经多次提到，在物理执行期间，RDD 会被分为一系列的分区，每个分区都是整个数据的子集。当 Spark 调度并运行任务时，Spark 会为每个分区中的数据创建出一个任务。该任务在默认情况下会需要集群中的一个计算核心来执行。Spark 也会针对 RDD 直接自动推断出合适的并行度，这对于大多数用例来说已经足够了。输入 RDD 一般会根据其底层的存储系统选择并行度。例如，从 HDFS 上读数据的输入 RDD 会为数据在 HDFS 上的每个文件区块创建一个分区。从数据混洗后的 RDD 派生下来的 RDD 则会采用与其父 RDD 相同的并行度。

并行度会从两方面影响程序的性能。首先，当并行度过低时，Spark 集群会出现资源闲置的情况。比如，假设你的应用有 1000 个可使用的计算核心，但所运行的步骤只有 30 个任务，你就应该提高并行度来充分利用更多的计算核心。而当并行度过高时，每个分区产生的间接开销累计起来就会更大。评判并行度是否过高的标准包括任务是否是几乎在瞬间（毫秒级）完成的，或者是否观察到任务没有读写任何数据。

Spark 提供了两种方法来对操作的并行度进行调优。第一种方法是在数据混洗操作时，使用参数的方式为混洗后的 RDD 指定并行度。第二种方法是对于任何已有的 RDD，可以进行重新分区来获取更多或者更少的分区数。重新分区操作通过 `repartition()` 实现，该操作会把 RDD 随机打乱并分成设定的分区数目。如果你确定要减少 RDD 分区，可以使用 `coalesce()` 操作。由于没有打乱数据，该操作比 `repartition()` 更为高效。如果你认为当前的并行度过高或者过低，可以利用这些方法对数据分布进行重新调整。

举个例子，假设我们从 S3 上读取了大量数据，然后马上进行 `filter()` 操作筛选掉数据集中的绝大部分数据。默认情况下，`filter()` 返回的 RDD 的分区数和其父节点一样，这样可能会产生很多空的分区或者只有很少数据的分区。在这样的情况下，可以通过合并得到分区更少的 RDD 来提高应用性能，如例 8-11 所示。

例 **8-11**：在 PySpark shell 中合并分区过多的 RDD

```
# 以可以匹配数千个文件的通配字符串作为输入
>>> input = sc.textFile("s3n://log-files/2014/*.log")
>>> input.getNumPartitions()
35154
# 排除掉大部分数据的筛选方法
>>> lines = input.filter(lambda line: line.startswith("2014-10-17"))
>>> lines.getNumPartitions()
```

```
35154
# 在缓存lines之前先对其进行合并操作
>>> lines = lines.coalesce(5).cache()
>>> lines.getNumPartitions()
4
# 可以在合并之后的RDD上进行后续分析
>>> lines.count()
```

8.4.2 序列化格式

当 Spark 需要通过网络传输数据，或是将数据溢写到磁盘上时，Spark 需要把数据序列化为二进制格式。序列化会在数据进行混洗操作时发生，此时有可能需要通过网络传输大量数据。默认情况下，Spark 会使用 Java 内建的序列化库。Spark 也支持使用第三方序列化库 Kryo（https://github.com/EsotericSoftware/kryo），可以提供比 Java 的序列化工具更短的序列化时间和更高压缩比的二进制表示，但不能直接序列化全部类型的对象。几乎所有的应用都在迁移到 Kryo 后获得了更好的性能。

要使用 Kryo 序列化工具，你需要设置 `spark.serializer` 为 `org.apache.spark.serializer.KryoSerializer`。为了获得最佳性能，你还应该向 Kryo 注册你想要序列化的类，如例 8-12 所示。注册类可以让 Kryo 避免把每个对象的完整的类名写下来，成千上万条记录累计节省的空间相当可观。如果你想强制要求这种注册，可以把 `spark.kryo.registrationRequired` 设置为 `true`，这样 Kryo 会在遇到未注册的类时抛出错误。

例 **8-12**：使用 Kryo 序列化工具并注册所需类

```
val conf = new SparkConf()
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
// 严格要求注册类
conf.set("spark.kryo.registrationRequired", "true")
conf.registerKryoClasses(Array(classOf[MyClass], classOf[MyOtherClass]))
```

不论是选用 Kryo 还是 Java 序列化，如果代码中引用到了一个没有扩展 Java 的 Serializable 接口的类，你都会遇到 `NotSerializableException`。在这种情况下，要查出引发问题的类是比较困难的，因为用户代码会引用到许许多多不同的类。很多 JVM 都支持通过一个特别的选项来帮助调试这一情况："`-Dsun.io.serialization.extended DebugInfo=true`”。你可以通过设置 `spark-submit` 的 `--driver-java-options` 和

`--executor-java-options` 标记来打开这个选项。一旦找到了有问题的类，最简单的解决方法就是把这个类改为实现了 Serializable 接口的形式。如果没有办法修改这个产生问题的类，你就需要采用一些高级的变通策略，比如为这个类创建一个子类并实现 Java 的 Externalizable 接口（https://docs.oracle.com/javase/7/docs/api/java/io/Externalizable.html），或者自定义 Kryo 的序列化行为。

8.4.3　内存管理

内存对 Spark 来说有几种不同的用途，理解并调优 Spark 的内存使用方法可以帮助优化 Spark 的应用。在各个执行器进程中，内存有以下所列几种用途。

- **RDD存储**

  当调用 RDD 的 `persist()` 或 `cache()` 方法时，这个 RDD 的分区会被存储到缓存区中。Spark 会根据 `spark.storage.memoryFraction` 限制用来缓存的内存占整个 JVM 堆空间的比例大小。如果超出限制，旧的分区数据会被移出内存。

- 数据混洗与聚合的缓存区

  当进行数据混洗操作时，Spark 会创建出一些中间缓存区来存储数据混洗的输出数据。这些缓存区用来存储聚合操作的中间结果，以及数据混洗操作中直接输出的部分缓存数据。Spark 会尝试根据 `spark.shuffle.memoryFraction` 限定这种缓存区内存占总内存的比例。

- 用户代码

  Spark 可以执行任意的用户代码，所以用户的函数可以自行申请大量内存。例如，如果一个用户应用分配了巨大的数组或者其他对象，那这些都会占用总的内存。用户代码可以访问 JVM 堆空间中除分配给 RDD 存储和数据混洗存储以外的全部剩余空间。

在默认情况下，Spark 会使用 60%的空间来存储 RDD，20% 存储数据混洗操作产生的数据，剩下的 20% 留给用户程序。用户可以自行调节这些选项来追求更好的性能表现。如果用户代码中分配了大量的对象，那么降低 RDD 存储和数据混洗存储所占用的空间可以有效避

免程序内存不足的情况。

除了调整内存各区域比例，我们还可以为一些工作负载改进缓存行为的某些要素。Spark 默认的 `cache()` 操作会以 `MEMORY_ONLY` 的存储等级持久化数据。这意味着如果缓存新的 nRDD 分区时空间不够，旧的分区就会直接被删除。当用到这些分区数据时，再进行重算。所以有时以 `MEMORY_AND_DISK` 的存储等级调用 `persist()` 方法会获得更好的效果，因为在这种存储等级下，内存中放不下的旧分区会被写入磁盘，当再次需要用到的时候再从磁盘上读取回来。这样的代价有可能比重算各分区要低很多，也可以带来更稳定的性能表现。当 RDD 分区的重算代价很大（比如从数据库中读取数据）时，这种设置尤其有用。可用存储等级的完整列表请参见表 3-6。

对于默认缓存策略的另一个改进是缓存序列化后的对象而非直接缓存。我们可以通过 `MEMORY_ONLY_SER` 或者 `MEMORY_AND_DISK_SER` 的存储等级来实现这一点。缓存序列化后的对象会使缓存过程变慢，因为序列化对象也会消耗一些代价，不过这可以显著减少 JVM 的垃圾回收时间，因为很多独立的记录现在可以作为单个序列化的缓存而存储。垃圾回收的代价与堆里的对象数目相关，而不是和数据的字节数相关。这种缓存方式会把大量对象序列化为一个巨大的缓存区对象。如果你需要以对象的形式缓存大量数据（比如数 GB 的数据），或者是注意到了长时间的垃圾回收暂停，可以考虑配置这个选项。这些暂停时间可以在应用用户界面中显示的每个任务的垃圾回收时间那一栏看到。

8.4.4　硬件供给

提供给 Spark 的硬件资源会显著影响应用的完成时间。影响集群规模的主要参数包括分配给每个执行器节点的内存大小、每个执行器节点占用的核心数、执行器节点总数，以及用来存储临时数据的本地磁盘数量。

在各种部署模式下，执行器节点的内存都可以通过 `spark.executor.memory` 配置项或者 `spark-submit` 的 `--executor-memory` 标记来设置。而执行器节点的数目以及每个执行器进程的核心数的配置选项则取决于各种部署模式。在 YARN 模式下，你可以通过 `spark.executor.cores` 或 `--executor-cores` 标记来设置执行器节点的核心数，通过 `--num-executors` 设置执行器节点的总数。而在 Mesos 和独立模式中，Spark 则会从调度器提供的资源中获取尽可能多的核心以用于执行器节点。不过，

Mesos 和独立模式也支持通过设置 `spark.cores.max` 项来限制一个应用中所有执行器节点所使用的核心总数。本地磁盘则用来在数据混洗操作中存储临时数据。

一般来说，更大的内存和更多的计算核心对 Spark 应用会更有用处。Spark 的架构允许线性伸缩；双倍的资源通常能使应用的运行时间减半。在调整集群规模时，需要额外考虑的方面还包括是否在计算中把中间结果数据集缓存起来。如果确实要使用缓存，那么内存中缓存的数据越多，应用的表现就会越好。Spark 用户界面中的存储页面会展示所缓存的数据中有哪些部分保留在内存中。你可以从在小集群上只缓存一部分数据开始，然后推算缓存大量数据所需要的总内存量。

除了内存和 CPU 核心，Spark 还要用到本地磁盘来存储数据混洗操作的中间数据，以及溢写到磁盘中的 RDD 分区数据。因此，使用大量的本地磁盘可以帮助提升 Spark 应用的性能。在 YARN 模式下，由于 YARN 提供了自己的指定临时数据存储目录的机制，Spark 的本地磁盘配置项会直接从 YARN 的配置中读取。而在独立模式下，我们可以在部署集群时，在 `spark-env.sh` 文件中设置环境变量 `SPARK_LOCAL_DIRS`，这样 Spark 应用启动时就会自动读取这个配置项的值。如果运行的是 Mesos 模式，或者是在别的模式下需要重载集群默认的存储位置时，可以使用 `spark.local.dir` 选项来实现配置。在所有情况下，本地目录的设置都应当使用由单个逗号隔开的目录列表。一般的做法是在磁盘的每个分卷中都为 Spark 设置一个本地目录。写操作会被均衡地分配到所有提供的目录中。磁盘越多，可以提供的总吞吐量就越高。

切记，“越多越好”的原则在设置执行器节点内存时并不一定适用。使用巨大的堆空间可能会导致垃圾回收的长时间暂停，从而严重影响 Spark 作业的吞吐量。有时，使用较小内存（比如不超过 64GB）的执行器实例可以缓解该问题。Mesos 和 YARN 本身就已经支持在同一个物理主机上运行多个较小的执行器实例，所以使用较小内存的执行器实例不代表应用所使用的总资源一定会减少。而在 Spark 的独立模式中，我们需要启动多个工作节点实例（使用 `SPARK_WORKER_INSTANCES` 指定）来让单个应用在一台主机上运行于多个执行器节点中。这样的限制很有可能会在以后的版本中被去掉。除了给单个执行器实例分配较小的内存，我们还可以用序列化的格式存储数据（参见 8.4.3 节）来减轻垃圾回收带来的影响。

8.5　总结

当你读完本章时，相信你已经为处理生产环境中的 Spark 用例作好了充分的准备。我们讲到了 Spark 的配置项管理、执行的组成部分、用户界面中的相关指标，以及对生产环境中的工作负载常用的调优技巧。要深入了解 Spark 调优，请访问官方文档中的调优指南（http://spark.apache.org/docs/latest/tuning.html）。

第 9 章　Spark SQL

本章介绍 Spark 用来操作结构化和半结构化数据的接口——Spark SQL。结构化数据是指任何有结构信息的数据。所谓结构信息，就是每条记录共用的已知的字段集合。当数据符合这样的条件时，Spark SQL 就会使得针对这些数据的读取和查询变得更加简单高效。具体来说，Spark SQL 提供了以下三大功能（见图 9-1）。

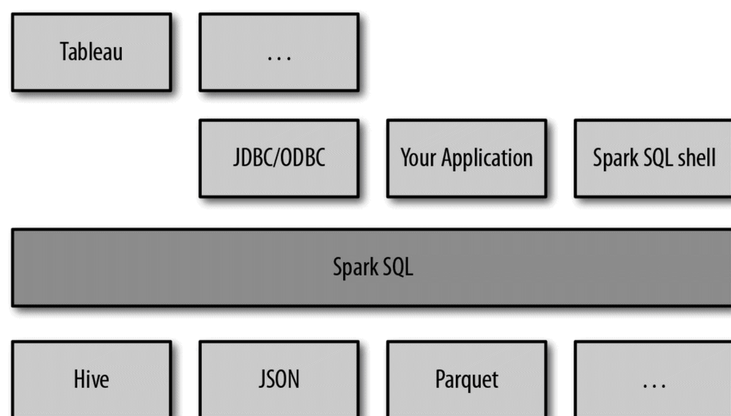


图 **9-1**：**Spark SQL** 的用途

(1) Spark SQL 可以从各种结构化数据源（例如 JSON、Hive、Parquet 等）中读取数据。

(2) Spark SQL 不仅支持在 Spark 程序内使用 SQL 语句进行数据查询，也支持从类似商业智能软件 Tableau 这样的外部工具中通过标准数据库连接器（JDBC/ODBC）连接 Spark SQL 进行查询。

(3) 当在 Spark 程序内使用 Spark SQL 时，Spark SQL 支持 SQL 与常规的 Python/Java/Scala 代码高度整合，包括连接 RDD 与 SQL 表、公开的自定义 SQL 函数接口等。这样一来，许多工作都更容易实现了。

为了实现这些功能，Spark SQL 提供了一种特殊的 RDD，叫作 SchemaRDD。1SchemaRDD 是存放 `Row` 对象的 RDD，每个 `Row` 对象代表一行记录。SchemaRDD 还包含记录的结构信息（即数据字

段）。SchemaRDD 看起来和普通的 RDD 很像，但是在内部，SchemaRDD 可以利用结构信息更加高效地存储数据。此外，SchemaRDD 还支持 RDD 上所没有的一些新操作，比如运行 SQL 查询。SchemaRDD 可以从外部数据源创建，也可以从查询结果或普通 RDD 中创建。

> 11.3.0 及后续版本中，SchemaRDD 已经被 DataFrame 所取代。——译者注

本章会先讲解如何在常规 Spark 程序中使用 SchemaRDD，以读取和查询结构化数据。接下来会讲解 Spark SQL 的 JDBC 服务器，它可以让你在一个共享的服务器上运行 Spark SQL，也可以让 SQL shell 或者类似 Tableau 的可视化工具连接它而使用。最后会讨论更多高级特性。Spark SQL 是 Spark 中比较新的组件，在 Spark 1.3 以及后续版本中还会有重大升级，因此要想获取关于 Spark SQL 和 SchemaRDD 的最新信息，请访问最新版本的文档。

在学习本章的过程中，我们会使用 Spark SQL 探索一个包含推文的 JSON 格式的文件。如果你手边没有现成的推文，你可以使用 Databricks 参考应用（http://databricks.gitbooks.io/databricks-spark-reference-applications/content/twitter_classifier/README.html）来下载一些。当然，你也可以直接使用本书 Git 仓库中的 files/testweet.json 文件。

9.1　连接Spark SQL

跟 Spark 的其他程序库一样，要在应用中引入 Spark SQL 需要添加一些额外的依赖。这种分离机制使得 Spark 内核的编译无需依赖大量额外的包。

Apache Hive 是 Hadoop 上的 SQL 引擎，Spark SQL 编译时可以包含 Hive 支持，也可以不包含。包含 Hive 支持的 Spark SQL 可以支持 Hive 表访问、UDF（用户自定义函数）、SerDe（序列化格式和反序列化格式），以及 Hive 查询语言（HiveQL/HQL）。需要强调的一点是，如果要在 Spark SQL 中包含 Hive 的库，并不需要事先安装 Hive。一般来说，最好还是在编译 Spark SQL 时引入 Hive 支持，这样就可以使用这些特性了。如果你下载的是二进制版本的 Spark，它应该已经在编译时添加了 Hive 支持。而如果你是从代码编译 Spark，你应该使用 `sbt/sbt -Phive assembly` 编译，以打开 Hive 支持。2

2sbt/sbt 命令已经被 build/sbt 命令替代。——译者注

如果你的应用与 Hive 之间发生了依赖冲突，并且无法通过依赖排除以及依赖封装解决问题，你也可以使用没有 Hive 支持的 Spark SQL 进行编译和连接。那样的话，你要连接的就是另一个 Maven 工件了。

在 Java 以及 Scala 中，连接带有 Hive 支持的 Spark SQL 的 Maven 索引如例 9-1 所示。

例 **9-1**：带有 Hive 支持的 Spark SQL 的 Maven 索引

```
groupId = org.apache.spark
artifactId = spark-hive_2.10
version = 1.2.0
```

如果你不能引入 Hive 依赖，那就应该使用工件 spark-sql_2.10 来代替 spark-hive_2.10。

跟其他的 Spark 程序库一样，在 Python 中不需要对构建方式进行任何修改。

当使用 Spark SQL 进行编程时，根据是否使用 Hive 支持，有两个不同的入口。推荐使用的入口是 HiveContext，它可以提供 HiveQL 以及其他依赖于 Hive 的功能的支持。更为基础的 SQLContext 则支持 Spark SQL 功能的一个子集，子集中去掉了需要依赖于 Hive 的功能。这种分离主要是为那些可能会因为引入 Hive 的全部依赖而陷入依赖冲突的用户而设计的。使用 HiveContext 不需要事先部署好 Hive。

我们推荐使用 HiveQL 作为 Spark SQL 的查询语言。关于 HiveQL 已经有许多资料面世，包括 *Programming Hive*（http://shop.oreilly.com/product/0636920023555.do）以及在线的 Hive 语言手册（https://cwiki.apache.org/confluence/display/Hive/LanguageManual）。在 Spark1.0 和 1.1 中，Spark SQL 是基于 Hive 0.12 的，而在 Spark 1.2 中，Spark SQL 支持 Hive 0.13。如果你了解标准 SQL，应该也会对 HiveQL 非常熟悉。



Spark SQL 是 Spark 中一个较新的组件，正在快速发展中。兼容的 Hive 版本会在将来不断变化，所

以请查阅最新版本的文档以获取详细信息。

最后，若要把 Spark SQL 连接到一个部署好的 Hive 上，你必须把 hive-site.xml 复制到 Spark 的配置文件目录中（$SPARK_HOME/conf）。即使没有部署好 Hive，Spark SQL 也可以运行。

需要注意的是，如果你没有部署好 Hive，Spark SQL 会在当前的工作目录中创建出自己的 Hive 元数据仓库，叫作 metastore_db。此外，如果你尝试使用 HiveQL 中的 `CREATE TABLE`（并非 `CREATE EXTERNAL TABLE`）语句来创建表，这些表会被放在你默认的文件系统中的 /user/hive/warehouse 目录中（如果你的 classpath 中有配好的 hdfs-site.xml，默认的文件系统就是 HDFS，否则就是本地文件系统）。

9.2 在应用中使用Spark SQL

Spark SQL 最强大之处就是可以在 Spark 应用内使用。这种方式让我们可以轻松读取数据并使用 SQL 查询，同时还能把这一过程和普通的 Python/Java/Scala 程序代码结合在一起。

要以这种方式使用 Spark SQL，需要基于已有的 SparkContext 创建出一个 HiveContext（如果使用的是去除了 Hive 支持的 Spark 版本，则创建出 SQLContext）。这个上下文环境提供了对 Spark SQL 的数据进行查询和交互的额外函数。使用 HiveContext 可以创建出表示结构化数据的 SchemaRDD，并且使用 SQL 或是类似 `map()` 的普通 RDD 操作来操作这些 SchemaRDD。

9.2.1 初始化Spark SQL

要开始使用 Spark SQL，首先要在程序中添加一些 import 声明，如例 9-2 所示。

例 **9-2**：Scala 中 SQL 的 import 声明

```
// 导入Spark SQL
import org.apache.spark.sql.hive.HiveContext
// 如果不能使用hive依赖的话
import org.apache.spark.sql.SQLContext
```

Scala 用户能应该注意到，这里没有用类似在导入 SparkContext 时的

方法那样导入 `HiveContext._` 来访问隐式转换。隐式转换被用来把带有类型信息的 RDD 转变为专门用于 Spark SQL 查询的 RDD（也就是 SchemaRDD）。在创建出 `HiveContext` 的实例之后，通过添加如例 9-3 所示的代码导入必要的隐式转换支持。Java 和 Python 版本的 import 声明分别如例 9-4 和例 9-5 所示。

例 **9-3**：Scala 中 SQL 需要导入的隐式转换支持

```
// 创建Spark SQL的HiveContext
val hiveCtx = ...
// 导入隐式转换支持
import hiveCtx._
```

例 **9-4**：Java 中 SQL 的 import 声明

```
// 导入Spark SQL
import org.apache.spark.sql.hive.HiveContext;
// 当不能使用hive依赖时
import org.apache.spark.sql.SQLContext;
// 导入JavaSchemaRDD
import org.apache.spark.sql.SchemaRDD;
import org.apache.spark.sql.Row;
```

例 **9-5**：Python 中 SQL 的 import 声明

```
# 导入Spark SQL
from pyspark.sql import HiveContext, Row
# 当不能引入hive依赖时
from pyspark.sql import SQLContext, Row
```

添加好 import 声明之后，需要创建出一个 HiveContext 对象。而如果无法引入 Hive 依赖，就创建出一个 SQLContext 对象作为 SQL 的上下文环境（如例 9-6 至例 9-8 所示）。这两个类都需要传入一个 SparkContext 对象作为运行的基础。

例 **9-6**：在 Scala 中创建 SQL 上下文环境

```
val sc = new SparkContext(...)
val hiveCtx = new HiveContext(sc)
```

例 **9-7**：在 Java 中创建 SQL 上下文环境

```
JavaSparkContext ctx = new JavaSparkContext(...);
SQLContext sqlCtx = new HiveContext(ctx);
```

例 **9-8**：在 Python 中创建 SQL 上下文环境

```
hiveCtx = HiveContext(sc)
```

有了 HiveContext 或者 SQLContext 之后，我们就可以准备读取数据并进行查询了。

9.2.2　基本查询示例

要在一张数据表上进行查询，需要调用 HiveContext 或 SQLContext 中的 `sql()` 方法。要做的第一件事就是告诉 Spark SQL 要查询的数据是什么。因此，需要先从 JSON 文件中读取一些推特数据，把这些数据注册为一张临时表并赋予该表一个名字，然后就可以用 SQL 来查询它了。（9.3 节会更深入地讲解读取数据的细节。）接下来，就可以根据 `retweetCount` 字段（转发计数）选出最热门的推文，如例 9-9 至例 9-11 所示。

例 **9-9**：在 Scala 中读取并查询推文

```
val input = hiveCtx.jsonFile(inputFile)
// 注册输入的SchemaRDD
input.registerTempTable("tweets")
// 依据retweetCount（转发计数）选出推文
val topTweets = hiveCtx.sql("SELECT text, retweetCount FROM
  tweets ORDER BY retweetCount LIMIT 10")
```

例 **9-10**：在 Java 中读取并查询推文

```
SchemaRDD input = hiveCtx.jsonFile(inputFile);
// 注册输入的SchemaRDD
input.registerTempTable("tweets");
// 依据retweetCount（转发计数）选出推文
SchemaRDD topTweets = hiveCtx.sql("SELECT text, retweetCount FROM
  tweets ORDER BY retweetCount LIMIT 10");
```

例 **9-11**：在 Python 中读取并查询推文

```
input = hiveCtx.jsonFile(inputFile)
# 注册输入的SchemaRDD
input.registerTempTable("tweets")
# 依据retweetCount（转发计数）选出推文
topTweets = hiveCtx.sql("""SELECT text, retweetCount FROM
  tweets ORDER BY retweetCount LIMIT 10""")
```



如果你已经有安装好的 Hive，并且已经把你的 hive-site.xml 文件复制到了 $SPARK_HOME/conf 目录下，那么你也可以直接运行 `hiveCtx.sql` 来查询已有的 Hive 表。

9.2.3 SchemaRDD

读取数据和执行查询都会返回 SchemaRDD。SchemaRDD 和传统数据库中的表的概念类似。从内部机理来看，SchemaRDD 是一个由 `Row` 对象组成的 RDD，附带包含每列数据类型的结构信息。`Row` 对象只是对基本数据类型（如整型和字符串型等）的数组的封装。我们会在下一部分中进一步探讨 `Row` 对象的细节。

需要特别注意的是，在今后的 Spark 版本中（1.3 及以后），SchemaRDD 这个名字可能会被改为 DataFrame。这一重命名举动在本书编写完成时仍在讨论中。

SchemaRDD 仍然是 RDD，所以你可以对其应用已有的 RDD 转化操作，比如 `map()` 和 `filter()`。然而，SchemaRDD 也提供了一些额外的功能支持。最重要的是，你可以把任意 SchemaRDD 注册为临时表，这样就可以使用 HiveContext.sql 或 SQLContext.sql 来对它进行查询了。你可以通过 SchemaRDD 的 `registerTempTable()` 方法这么做，如例 9-9 到例 9-11 所示。



临时表是当前使用的 HiveContext 或 SQLContext 中的临时变量，在你的应用退出时这些临时表就不再存在了。

SchemaRDD 可以存储一些基本数据类型，也可以存储由这些类型组成的结构体和数组。

SchemaRDD 使用 HiveQL 语法（https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL）定义的类型。表 9-1 列出了支持的数据类型。[3]

3编译时除通过 `-Phive` 打开 Hive 支持外，还需打开 `-Phive-thriftserver` 选项。——译者注

表9-1：SchemaRDD中可以存储的数据类型

Spark SQL类型	Scala类型	Java类型	Python
TINYINT	Byte	Byte/byte	int/long（在 -128 到 127 之间）
SMALLINT	Short	Short/short	int/long（在 -32768 到 32767 之间）
INT	Int	Int/int	int 或 long
BIGINT	Long	Long/long	long
FLOAT	Float	Float/float	float
DOUBLE	Double	Double/double	float
DECIMAL	Scala.math.BigDecimal	java.math.BigDecimal	decimal.Decimal
STRING	String	String	string
BINARY	Array[Byte]	byte[]	bytearray
BOOLEAN	Boolean	Boolean/boolean	bool
TIMESTAMP	java.sql.TimeStamp	java.sql.TimeStamp	datetime.datetime
ARRAY<DATA_TYPE>	Seq	List	list、tuple 或 array
MAP<KEY_TYPE, VAL_TYPE>	Map	Map	dict
STRUCT<COL1: COL1_TYPE, ...>	Row	Row	Row

最后一种类型，也就是结构体，在 Spark SQL 中直接被表示为其他的 Row 对象。所有这些复杂类型都可以互相嵌套。比如，你可以有结构体组成的数组，或包含结构体的映射表。

使用Row对象

`Row` 对象表示 SchemaRDD 中的记录，其本质就是一个定长的字段数组。在 Scala/Java 中，`Row` 对象有一系列 getter 方法，可以通过下标获取每个字段的值。标准的取值方法 `get`（或 Scala 中的 `apply`），读入一个列的序号然后返回一个 `Object` 类型（或 Scala 中的 `Any` 类型）的对象，然后由我们把对象转为正确的类型。对于 `Boolean`、`Byte`、`Double`、`Float`、`Int`、`Long`、`Short` 和 `String` 类型，都有对应的 `getType()` 方法，可以把值直接作为相应的类型返回。例如，`getString(0)` 会把字段 0 的值作为字符串返回，如例 9-12 和例 9-13 所示。

例 **9-12**：在 Scala 中访问 topTweet 这个 SchemaRDD 中的 text 列（也就是第一列）

```
val topTweetText = topTweets.map(row => row.getString(0))
```

例 **9-13**：在 Java 中访问 topTweet 这个 SchemaRDD 中的 text 列（也就是第一列）

```
JavaRDD<String> topTweetText = topTweets.toJavaRDD().map(new Function<Row,
    public String call(Row row) {
      return row.getString(0);
    }});
```

在 Python 中，由于没有显式的类型系统，`Row` 对象变得稍有不同。我们使用 `row[i]` 来访问第 *i* 个元素。除此之外，Python 中的 `Row` 还支持以 `row.`*`column_name`* 的形式使用名字来访问其中的字段，如例 9-14 所示。如果你不确定具体的列名，我们会在 9.3.3 节中讲到如何输出结构信息。

例 **9-14**：在 Python 中访问 topTweet 这个 SchemaRDD 中的 text 列

```
topTweetText = topTweets.map(lambda row: row.text)
```

9.2.4　缓存

Spark SQL 的缓存机制与 Spark 中的稍有不同。由于我们知道每个列的类型信息，所以 Spark 可以更加高效地存储数据。为了确保使用更节约内存的表示方式进行缓存而不是储存整个对象，应当使用专门的 `hiveCtx.cacheTable("tableName")` 方法。当缓存数据表时，Spark SQL 使用一种列式存储格式在内存中表示数据。这些缓存下来的表只会在驱动器程序的生命周期里保留在内存中，所以如果驱动器进程退出，就需要重新缓存数据。和缓存 RDD 时的动机一样，如果想在同样的数据上多次运行任务或查询时，就应把这些数据表缓存起来。



在 Spark 1.2 中，RDD 上原有的 `cache()` 方法也会引发一次对 `cacheTable()` 方法的调用。

你也可以使用 HiveQL/SQL 语句来缓存表。只需要运行 `CACHE TABLE`*`tableName`* 或 `UNCACHE TABLE`*`tableName`* 来缓存表或者删除已有的缓存即可。这种使用方式在 JDBC 服务器的命令行客户端中很常用。

被缓存的 SchemaRDD 以与其他 RDD 相似的方式在 Spark 的应用用

户界面中呈现，如图 9-2 所示。

图 **9-2**：**Spark SQL** 的 **SchemaRDD** 用户界面

我们会在 9.6 节讨论 Spark SQL 中的缓存机制对性能的影响。

9.3　读取和存储数据

Spark SQL 支持很多种结构化数据源，可以让你跳过复杂的读取过程，轻松从各种数据源中读取到 `Row` 对象。这些数据源包括 Hive 表、JSON 和 Parquet 文件。此外，当你使用 SQL 查询这些数据源中的数据并且只用到了一部分字段时，Spark SQL 可以智能地只扫描这些用到的字段，而不是像 `SparkContext.hadoopFile` 中那样简单粗暴地扫描全部数据。

除这些数据源之外，你也可以在程序中通过指定结构信息，将常规的 RDD 转化为 SchemaRDD。这使得在 Python 或者 Java 对象上运行 SQL 查询更加简单。当需要计算许多数值时，SQL 查询往往更加简洁（比如要同时求出平均年龄、最大年龄、不重复的用户 ID 数目等）。不仅如此，你还可以自如地将这些 RDD 和来自其他 Spark SQL 数据源的 SchemaRDD 进行连接操作。在本节中，我们会讲解外部数据源以及这种使用 RDD 的方式。

9.3.1　Apache Hive

当从 Hive 中读取数据时，Spark SQL 支持任何 Hive 支持的存储格式（SerDe），包括文本文件、RCFiles、ORC、Parquet、Avro，以及

Protocol Buffer。

要把 Spark SQL 连接到已经部署好的 Hive 上，你需要提供一份 Hive 配置。你只需要把你的 hive-site.xml 文件复制到 Spark 的 ./conf/ 目录下即可。如果你只是想探索一下 Spark SQL 而没有配置 hive-site.xml 文件，那么 Spark SQL 则会使用本地的 Hive 元数据仓，并且同样可以轻松地将数据读取到 Hive 表中进行查询。

例 9-15 至例 9-17 展示了如何查询一张 Hive 表。Hive 示例表有两列，分别是 key（一个整型值）和 value（一个字符串）。我们会在本章稍后介绍如何创建这样的表。

例 **9-15**：使用 Python 从 Hive 读取

```
from pyspark.sql import HiveContext

hiveCtx = HiveContext(sc)
rows = hiveCtx.sql("SELECT key, value FROM mytable")
keys = rows.map(lambda row: row[0])
```

例 **9-16**：使用 Scala 从 Hive 读取

```
import org.apache.spark.sql.hive.HiveContext

val hiveCtx = new HiveContext(sc)
val rows = hiveCtx.sql("SELECT key, value FROM mytable")
val keys = rows.map(row => row.getInt(0))
```

例 **9-17**：使用 Java 从 Hive 读取

```
import org.apache.spark.sql.hive.HiveContext;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.SchemaRDD;
HiveContext hiveCtx = new HiveContext(sc);
SchemaRDD rows = hiveCtx.sql("SELECT key, value FROM mytable");
JavaRDD<Integer> keys = rdd.toJavaRDD().map(new Function<Row, Integer>()
  public Integer call(Row row) { return row.getInt(0); }
});
```

9.3.2 Parquet

Parquet（http://parquet.apache.org/）是一种流行的列式存储格

式，可以高效地存储具有嵌套字段的记录。Parquet 格式经常在 Hadoop 生态圈中被使用，它也支持 Spark SQL 的全部数据类型。Spark SQL 提供了直接读取和存储 Parquet 格式文件的方法。

首先，你可以通过 `HiveContext.parquetFile` 或者 `SQLContext.parquetFile` 来读取数据，如例 9-18 所示。

例 **9-18**：Python 中的 Parquet 数据读取

```
# 从一个有name和favouriteAnimal字段的Parquet文件中读取数据
rows = hiveCtx.parquetFile(parquetFile)
names = rows.map(lambda row: row.name)
print "Everyone"
print names.collect()
```

你也可以把 Parquet 文件注册为 Spark SQL 的临时表，并在这张表上运行查询语句。在例 9-18 中我们读取了数据，接下来可以参照例 9-19 所示的对数据进行查询。

例 **9-19**：Python 中的 Parquet 数据查询

```
# 寻找熊猫爱好者
tbl = rows.registerTempTable("people")
pandaFriends = hiveCtx.sql("SELECT name FROM people WHERE favouriteAnimal = \"panda\"")
print "Panda friends"
print pandaFriends.map(lambda row: row.name).collect()
```

最后，你可以使用 `saveAsParquetFile()` 把 SchemaRDD 的内容以 Parquet 格式保存，如例 9-20 所示。

例 **9-20**：Python 中的 Parquet 文件保存

```
pandaFriends.saveAsParquetFile("hdfs://...")
```

9.3.3　JSON

如果你有一个 JSON 文件，其中的记录遵循同样的结构信息，那么 Spark SQL 就可以通过扫描文件推测出结构信息，并且让你可以使用名字访问对应字段（如例 9-21 所示）。如果你在一个包含大量 JSON 文件的目录中进行尝试，你就会发现 Spark SQL 的结构信息推断可以

让你非常高效地操作数据，而无需编写专门的代码来读取不同结构的文件。

要读取 JSON 数据，只要调用 `hiveCtx` 中的 `jsonFile()` 方法即可，如例 9-22 至例 9-24 所示。如果你想获得从数据中推断出来的结构信息，可以在生成的 SchemaRDD 上调用 `printSchema` 方法（见例 9-25）。

例 **9-21**：输入记录

```
{"name": "Holden"}
{"name": "Sparky The Bear", "lovesPandas":true,"knows": {"friends":["holde
```

例 **9-22**：在 Python 中使用 Spark SQL 读取 JSON 数据

```
input = hiveCtx.jsonFile(inputFile)
```

例 **9-23**：在 Scala 中使用 Spark SQL 读取 JSON 数据

```
val input = hiveCtx.jsonFile(inputFile)
```

例 **9-24**：在 Java 中使用 Spark SQL 读取 JSON 数据

```
SchemaRDD input = hiveCtx.jsonFile(jsonFile);
```

例 **9-25**：`printSchema()` 输出的结构信息

```
root
 |-- knows: struct (nullable = true)
 |    |-- friends: array (nullable = true)
 |    |    |-- element: string (containsNull = false)
 |-- lovesPandas: boolean (nullable = true)
 |-- name: string (nullable = true)
```

你可以在例 9-26 中看到以某些推文生成的结构信息。

例 9-26：推文的部分结构

```
root
 |-- contributorsIDs: array (nullable = true)
 |    |-- element: string (containsNull = false)
 |-- createdAt: string (nullable = true)
 |-- currentUserRetweetId: integer (nullable = true)
 |-- hashtagEntities: array (nullable = true)
 |    |-- element: struct (containsNull = false)
 |    |    |-- end: integer (nullable = true)
 |    |    |-- start: integer (nullable = true)
 |    |    |-- text: string (nullable = true)
 |-- id: long (nullable = true)
 |-- inReplyToScreenName: string (nullable = true)
 |-- inReplyToStatusId: long (nullable = true)
 |-- inReplyToUserId: long (nullable = true)
 |-- isFavorited: boolean (nullable = true)
 |-- isPossiblySensitive: boolean (nullable = true)
 |-- isTruncated: boolean (nullable = true)
 |-- mediaEntities: array (nullable = true)
 |    |-- element: struct (containsNull = false)
 |    |    |-- displayURL: string (nullable = true)
 |    |    |-- end: integer (nullable = true)
 |    |    |-- expandedURL: string (nullable = true)
 |    |    |-- id: long (nullable = true)
 |    |    |-- mediaURL: string (nullable = true)
 |    |    |-- mediaURLHttps: string (nullable = true)
 |    |    |-- sizes: struct (nullable = true)
 |    |    |    |-- 0: struct (nullable = true)
 |    |    |    |    |-- height: integer (nullable = true)
 |    |    |    |    |-- resize: integer (nullable = true)
 |    |    |    |    |-- width: integer (nullable = true)
 |    |    |    |-- 1: struct (nullable = true)
 |    |    |    |    |-- height: integer (nullable = true)
 |    |    |    |    |-- resize: integer (nullable = true)
 |    |    |    |    |-- width: integer (nullable = true)
 |    |    |    |-- 2: struct (nullable = true)
 |    |    |    |    |-- height: integer (nullable = true)
 |    |    |    |    |-- resize: integer (nullable = true)
 |    |    |    |    |-- width: integer (nullable = true)
 |    |    |    |-- 3: struct (nullable = true)
 |    |    |    |    |-- height: integer (nullable = true)
 |    |    |    |    |-- resize: integer (nullable = true)
 |    |    |    |    |-- width: integer (nullable = true)
 |    |    |-- start: integer (nullable = true)
 |    |    |-- type: string (nullable = true)
 |    |    |-- url: string (nullable = true)
 |-- retweetCount: integer (nullable = true)
```

看到这样的结构，我们会自然而然地想到如何访问嵌套字段和数组字段这个问题。如果你使用 Python，或已经把数据注册为了一张 SQL

表，你可以通过 . 来访问各个嵌套层级的嵌套元素（比如 *toplevel.nextlevel*）。而在 SQL 中可以通过用 [*element*] 指定下标来访问数组中的元素，如例 9-27 所示。

例 **9-27**：用 SQL 查询嵌套数据以及数组元素

```
select hashtagEntities[0].text from tweets LIMIT 1;
```

9.3.4　基于RDD

除了读取数据，也可以基于 RDD 创建 SchemaRDD。在 Scala 中，带有 case class 的 RDD 可以隐式转换成 SchemaRDD。

在 Python 中，可以创建一个由 `Row` 对象组成的 RDD，然后调用 `inferSchema()`，如例 9-28 所示。

例 **9-28**：在 Python 中使用 `Row` 和具名元组创建 SchemaRDD

```
happyPeopleRDD = sc.parallelize([Row(name="holden", favouriteBeverage="cof
happyPeopleSchemaRDD = hiveCtx.inferSchema(happyPeopleRDD)
happyPeopleSchemaRDD.registerTempTable("happy_people")
```

使用 Scala 的话，我们的老朋友隐式转换会帮我们处理好结构信息的推断（见例 9-29）。

例 **9-29**：在 Scala 中基于 case class 创建 SchemaRDD

```
case class HappyPerson(handle: String, favouriteBeverage: String)
...
// 创建了一个人的对象，并且把它转成SchemaRDD
val happyPeopleRDD = sc.parallelize(List(HappyPerson("holden", "coffee")))
// 注意：此处发生了隐式转换
// 该转换等价于sqlCtx.createSchemaRDD(happyPeopleRDD)
happyPeopleRDD.registerTempTable("happy_people")
```

在 Java 中，可以调用 `applySchema()` 把 RDD 转为 SchemaRDD，只需要这个 RDD 中的数据类型带有公有的 getter 和 setter 方法，并且可以被序列化，如例 9-30 所示。

例 **9-30**：在 Java 中基于 JavaBean 创建 SchemaRDD

```
class HappyPerson implements Serializable {
  private String name;
  private String favouriteBeverage;
  public HappyPerson() {}
  public HappyPerson(String n, String b) {
    name = n; favouriteBeverage = b;
  }
  public String getName() { return name; }
  public void setName(String n) { name = n; }
  public String getFavouriteBeverage() { return favouriteBeverage; }
  public void setFavouriteBeverage(String b) { favouriteBeverage = b; }
};
...
ArrayList<HappyPerson> peopleList = new ArrayList<HappyPerson>();
peopleList.add(new HappyPerson("holden", "coffee"));
JavaRDD<HappyPerson> happyPeopleRDD = sc.parallelize(peopleList);
SchemaRDD happyPeopleSchemaRDD = hiveCtx.applySchema(happyPeopleRDD,
  HappyPerson.class);
happyPeopleSchemaRDD.registerTempTable("happy_people");
```

9.4　JDBC/ODBC服务器

Spark SQL 也提供 JDBC 连接支持，这对于让商业智能（BI）工具连接到 Spark 集群上以及在多用户间共享一个集群的场景都非常有用。JDBC 服务器作为一个独立的 Spark 驱动器程序运行，可以在多用户之间共享。任意一个客户端都可以在内存中缓存数据表，对表进行查询。集群的资源以及缓存数据都在所有用户之间共享。

Spark SQL 的 JDBC 服务器与 Hive 中的 HiveServer2 相一致。由于使用了 Thrift 通信协议，它也被称为“Thrift server”。注意，JDBC 服务器支持需要 Spark 在打开 Hive 支持的选项下编译。4

> 4codegen 打开时，查询有可能会变慢，因为 Spark SQL 需要动态分析并编译代码，因此，短作业并不能真正体现 codegen 所带来的性能提升。——译者注

服务器可以通过 Spark 目录中的 `sbin/start-thriftserver.sh` 启动（见例 9-31）。这个脚本接受的参数选项大多与 `spark-submit` 相同（见 7.3 节）。默认情况下，服务器会在 `localhost:10000` 上进行监听，我们可以通过环境变量（`HIVE_SERVER2_THRIFT_PORT` 和

HIVE_SERVER2T_HRIFT_BIND_HOST）修改这些设置，也可以通过 Hive 配置选项（hive.server2.thrift.port 和 hive.server2.thrift.bind.host）来修改。你也可以通过命令行参数 --hiveconf property=value 来设置 Hive 选项。

例 **9-31**：启动 JDBC 服务器

```
./sbin/start-thriftserver.sh --master sparkMaster
```

Spark 也自带了 Beeline 客户端程序，我们可以使用它连接 JDBC 服务器，如例 9-32 和图 9-3 所示。这个简易的 SQL shell 可以让我们在服务器上运行命令。

例 **9-32**：使用 Beeline 连接 JDBC 服务器

```
holden@hmbp2:~/repos/spark$ ./bin/beeline -u jdbc:hive2://localhost:10000
Spark assembly has been built with Hive, including Datanucleus jars on cla
scan complete in 1ms
Connecting to jdbc:hive2://localhost:10000
Connected to: Spark SQL (version 1.2.0-SNAPSHOT)
Driver: spark-assembly (version 1.2.0-SNAPSHOT)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 1.2.0-SNAPSHOT by Apache Hive
0: jdbc:hive2://localhost:10000> show tables;
+----------+
| result   |
+----------+
| pokes    |
+----------+
1 row selected (1.182 seconds)
0: jdbc:hive2://localhost:10000>
```

```
holden@hmbp2: ~/repos/spark
holden@hmbp2: ~/repos/1230000000573        holden@hmbp2: ~/repos/spark
derby.log          LICENSE                 README.md          yarn
dev                logs                    repl
docker             make-distribution.sh    sbin
docs               metastore_db            sbt
holden@hmbp2:~/repos/spark$ ./sbin/start-thriftserver.sh --master local[*]
starting org.apache.spark.sql.hive.thriftserver.HiveThriftServer2, logging to /h
ome/holden/repos/spark/sbin/../logs/spark-holden-org.apache.spark.sql.hive.thrif
tserver.HiveThriftServer2-1-hmbp2.out
holden@hmbp2:~/repos/spark$ ./bin/beeline -u jdbc:hive2://localhost:10000
Spark assembly has been built with Hive, including Datanucleus jars on classpath
scan complete in 1ms
Connecting to jdbc:hive2://localhost:10000
Connected to: Spark SQL (version 1.2.0-SNAPSHOT)
Driver: spark-assembly (version 1.2.0-SNAPSHOT)
Transaction isolation: TRANSACTION_REPEATABLE_READ
Beeline version 1.2.0-SNAPSHOT by Apache Hive
0: jdbc:hive2://localhost:10000> show tables;
+---------+
| result  |
+---------+
| pokes   |
+---------+
1 row selected (1.473 seconds)
0: jdbc:hive2://localhost:10000>
```

图 **9-3**：启动 **JDBC** 服务器并使用 **Beeline** 客户端连接

> 当启动 JDBC 服务器时，JDBC 服务器会在后台运行并且将所有输出重定向到一个日志文件中。如果你在使用 JDBC 服务器进行查询的过程中遇到了问题，可以查看日志寻找更为完整的报错信息。

许多外部工具也可以通过 ODBC 连接 Spark SQL。Spark SQL 的 ODBC 驱动由 Simba（http://www.simba.com/）制作，可以从很多 Spark 供应商处下载到（比如 DataBricks Cloud、Datastax 以及 MapR）。它经常会被像 Microstrategy 或 Tableau 这样的商务智能工具所用到；你可以查一查如何把你的工具连接到 Spark SQL 上。由于 Spark SQL 使用了和 Hive 相同的查询语言以及服务器，大多数可以连接到 Hive 的商务智能工具也可以通过已有的 Hive 连接器来连接到 Spark SQL 上。

9.4.1　使用Beeline

在 Beeline 客户端中，你可以使用标准的 HiveQL 命令来创建、列举以及查询数据表。你可以从 Hive 语言手册（https://cwiki.apache.org/confluence/display/Hive/LanguageManual）中找到关于 HiveQL 的所有语法细节，这里只展示一些常见的操作。

首先，要从本地数据创建一张数据表，可以使用 `CREATE TABLE` 命令。然后使用 `LOAD DATA` 命令进行数据读取。Hive 支持读取带有固定分隔符的文本文件，比如 CSV 等格式的文件，如例 9-33 所示。

例 **9-33**：读取数据表

```
> CREATE TABLE IF NOT EXISTS mytable (key INT, value STRING)
  ROW FORMAT DELIMITED FIELDS TERMINATED BY ',';
> LOAD DATA LOCAL INPATH 'learning-spark-examples/files/int_string.csv'
INTO TABLE mytable;
```

要列举数据表，可以使用 `SHOW TABLES` 语句（如例 9-34 所示）。你也可以通过 `DESCRIBE` *`tableName`* 查看每张表的结构信息。

例 **9-34**：列举数据表

```
> SHOW TABLES;
mytable
Time taken: 0.052 seconds
```

如果你想要缓存数据表，使用 `CACHE TABLE` *`tableName`* 语句。缓存之后你可以使用 `UNCACHE TABLE` *`tableName`* 命令取消对表的缓存。需要注意的是，之前也提到过，缓存的表会在这个 JDBC 服务器上的所有客户端之间共享。

最后，在 Beeline 中查看查询计划很简单，对查询语句运行 `EXPLAIN` 即可，如例 9-35 所示。

例 **9-35**：Spark SQL shell 执行 EXPLAIN

```
spark-sql> EXPLAIN SELECT * FROM mytable where key = 1;
== Physical Plan ==
Filter (key#16 = 1)
 HiveTableScan [key#16,value#17], (MetastoreRelation default, mytable, Non
Time taken: 0.551 seconds
```

对于这个查询计划来说，Spark SQL 在一个 `HiveTableScan` 节点上使用了筛选操作。

在这里，你也可以直接写 SQL 语句对数据进行查询。Beeline shell 对于在多用户间共享的缓存数据表上进行快速的数据探索是非常有用的。

9.4.2　长生命周期的表与查询

使用 Spark SQL 的 JDBC 服务器的优点之一就是我们可以在多个不同程序之间共享缓存下来的数据表。JDBC Thrift 服务器是一个单驱动器程序，这就使得共享成为了可能。如前一节中所述，你只需要注册该数据表并对其运行 `CACHE` 命令，就可以利用缓存了。



Spark SQL 独立 shell

除了 JDBC 服务器，Spark SQL 也支持一个可以作为单独的进程使用的简易 shell，可以通过 ./bin/spark-sql 启动。这个 shell 会连接到你设置在 conf/hive-site.xml 中的 Hive 的元数据仓。如果不存在这样的元数据仓，Spark SQL 也会在本地新建一个。这个脚本主要对于本地开发比较有用。在共享的集群上，你应该使用 JDBC 服务器，让各用户通过 `beeline` 进行连接。

9.5　用户自定义函数

用户自定义函数，也叫 UDF，可以让我们使用 Python/Java/Scala 注册自定义函数，并在 SQL 中调用。这种方法很常用，通常用来给机构内的 SQL 用户们提供高级功能支持，这样这些用户就可以直接调用注册的函数而无需自己去通过编程来实现了。在 Spark SQL 中，编写 UDF 尤为简单。Spark SQL 不仅有自己的 UDF 接口，也支持已有的 Apache Hive UDF。

9.5.1　Spark SQL UDF

我们可以使用 Spark 支持的编程语言编写好函数，然后通过 Spark SQL 内建的方法传递进来，非常便捷地注册我们自己的 UDF。在 Scala 和 Python 中，可以利用语言原生的函数和 lambda 语法的支持，而在 Java 中，则需要扩展对应的 UDF 类。UDF 能够支持各种数据类型，返回类型也可以与调用时的参数类型完全不一样。

在 Python 和 Java 中，还需要用表 9-1 中列出的 SchemaRDD 对应的类型来指定返回值类型。Java 中的对应类型可以在 `org.apache.spark.sql.api.java.DataType` 中找到，而在 Python 中则需要导入 `DataType` 支持。

在例 9-36 和例 9-37 中，我们可以看到一个用来计算字符串长度的非常简易的 UDF，可以用它来计算推文的长度。

例 **9-36**：Python 版本的字符串长度 UDF

```
# 写一个求字符串长度的UDF
hiveCtx.registerFunction("strLenPython", lambda x: len(x), IntegerType())
lengthSchemaRDD = hiveCtx.sql("SELECT strLenPython('text') FROM tweets LIM
```

例 **9-37**：Scala 版本的字符串长度 UDF

```
registerFunction("strLenScala", (_: String).length)
val tweetLength = hiveCtx.sql("SELECT strLenScala('tweet') FROM tweets LIM
```

在 Java 中定义 UDF 需要一些额外的 import 声明。和在定义 RDD 函数时一样，根据我们要实现的 UDF 的参数个数，需要扩展特定的类，如例 9-38 和例 9-39 所示。

例 **9-38**：Java UDF import 声明

```
// 导入UDF函数类以及数据类型
// 注意：这些import路径可能会在将来的发行版中改变
import org.apache.spark.sql.api.java.UDF1;
import org.apache.spark.sql.types.DataTypes;
```

例 **9-39**：Java 版本的字符串长度 UDF

```
hiveCtx.udf().register("stringLengthJava", new UDF1<String, Integer>() {
    @Override
      public Integer call(String str) throws Exception {
      return str.length();
    }
}, DataTypes.IntegerType);
SchemaRDD tweetLength = hiveCtx.sql(
  "SELECT stringLengthJava('text') FROM tweets LIMIT 10");
List<Row> lengths = tweetLength.collect();
for (Row row : result) {
  System.out.println(row.get(0));
}
```

9.5.2 Hive UDF

Spark SQL 也支持已有的 Hive UDF。标准的 Hive UDF 已经自动包含在了 Spark SQL 中。如果需要支持自定义的 Hive UDF，我们要确保该 UDF 所在的 JAR 包已经包含在了应用中。需要注意的是，如果使用的是 JDBC 服务器，也可以使用 `--jars` 命令行标记来添加 JAR。开发 Hive UDF 不在本书的讨论范围之中，所以我们只会介绍一下如何使用已有的 Hive UDF。

要使用 Hive UDF，应该使用 HiveContext，而不能使用常规的 SQLContext。要注册一个 Hive UDF，只需调用 `hiveCtx.sql("CREATE TEMPORARY FUNCTION name AS class.function")`。

9.6 Spark SQL性能

就像本章开头所说的那样，Spark SQL 提供的高级查询语言及附加的类型信息可以使 Spark SQL 数据查询更加高效。

Spark SQL 不仅是给熟悉 SQL 的用户使用的。Spark SQL 使有条件的聚合操作变得非常容易，比如对多个列进行求值（如例 9-40 所示）。利用 Spark SQL 则不再需要像第 6 章中讨论的那样创建一些特殊的对象来进行这种操作。

例 **9-40**：Spark SQL 多列求和

```
SELECT SUM(user.favouritesCount), SUM(retweetCount), user.id FROM tweets
  GROUP BY user.id
```

Spark SQL 可以利用其对类型的了解来高效地表示数据。当缓存数据时，Spark SQL 使用内存式的列式存储。这不仅仅节约了缓存的空间，而且尽可能地减少了后续查询中针对某几个字段查询时的数据读取。

谓词下推可以让 Spark SQL 将查询中的一些部分工作“下移”到查询引擎上。如果我们只需在 Spark 中读取某些特定的记录，标准的方法是读入整个数据集，然后在上面执行筛选条件。然而，在 Spark SQL 中，如果底层的数据存储支持只读取键值在一个范围内的记录，或是其他某些限制条件，Spark SQL 就可以把查询语句中的筛选限制条件推到数据存储层，从而大大减少需要读取的数据。

性能调优选项

Spark SQL 的性能调优选项有很多，见表 9-2 所列。

表9-2：Spark SQL中的性能选项

选项	默认值	用途
spark.sql.codegen		设为 true 时，Spark SQL 会把每条查询语句在运行时编译为 Java 二进制代码。这可以提高大型查询的性能，但在进行小规模查询时会变慢
spark.sql.inMemoryColumnarStorage.compressed		自动对内存中的列式存储进行压缩
spark.sql.inMemoryColumnarStorage.batchSize		列式缓存时的每个批处理的大小。把这个值调大可能会导致内存不够的异常
spark.sql.parquet.compression.codec		使用哪种压缩编码器。可选的选项包括 uncompressed/snappy/gzip/lzo

使用 JDBC 连接接口和 Beeline shell 时，可以通过 `set` 命令设置包括这些性能选项在内的各种选项，如例 9-41 所示。

例 **9-41**：打开 `codegen` 选项的 Beeline 命令

```
beeline> set spark.sql.codegen=true;
SET spark.sql.codegen=true
spark.sql.codegen=true
Time taken: 1.196 seconds
```

在一个传统的 Spark SQL 应用中，可以在 Spark 配置中设置这些 Spark 属性，如例 9-42 所示。

例 **9-42**：在 Scala 中打开 `codegen` 选项的代码

```
conf.set("spark.sql.codegen", "true")
```

一些选项的配置需要给予特别的考量。第一个是 `spark.sql.codegen`，这个选项可以让 Spark SQL 把每条查询语句在运行前编译为 Java 二进制代码。由于生成了专门运行指定查询的代码，codegen 可以让大型查询或者频繁重复的查询明显变快。然而，在运行特别快（1 ～ 2 秒）的即时查询语句时，codegen 有可能会增加额外开销，因为 codegen 需要让每条查询走一遍编译的过程。5codegen 还是一个试验性的功能，但是我们推荐在所有大型的或者是重复运行的查询中使用 codegen。

5注意，codegen 打开时最开始的几条查询会格外慢，因为 Spark SQL 需要初始化它的编

> 译器。所以在测试 codegen 的额外开销之前你应该先运行 4 ~ 5 条查询语句。

调优时可能需要考虑的第二个选项是 `spark.sql.inMemoryColumnarStorage.batchSize`。在缓存 SchemaRDD 时，Spark SQL 会按照这个选项制定的大小（默认值是 1000）把记录分组，然后分批压缩。太小的批处理大小会导致压缩比过低，而批处理大小过大的话，比如当每个批次处理的数据超过内存所能容纳的大小时，也有可能会引发问题。如果你表中的记录比较大（包含数百个字段或者包含像网页这样非常大的字符串字段），你就可能需要调低批处理大小来避免内存不够（OOM）的错误。如果不是在这样的场景下，默认的批处理大小是比较合适的，因为压缩超过 1000 条记录时也基本无法获得更高的压缩比了。

9.7 总结

现在，我们学完了 Spark 利用 Spark SQL 进行结构化和半结构化数据处理的方式。除了本章探索过的查询语句，第 3 章到第 6 章中讲到的操作 RDD 的方法同样适用于 Spark SQL 中的 SchemaRDD。很多时候，我们会把 SQL 与其他的编程语言结合起来使用，以充分利用 SQL 的简洁性和编程语言擅长表达复杂逻辑的优点。而在使用 Spark SQL 时，Spark 执行引擎也能根据数据的结构信息对查询进行优化，让我们从中获益。

第 10 章　Spark Streaming

许多应用需要即时处理收到的数据，例如用来实时追踪页面访问统计的应用、训练机器学习模型的应用，还有自动检测异常的应用。Spark Streaming 是 Spark 为这些应用而设计的模型。它允许用户使用一套和批处理非常接近的 API 来编写流式计算应用，这样就可以大量重用批处理应用的技术甚至代码。

和 Spark 基于 RDD 的概念很相似，Spark Streaming 使用离散化流（discretized stream）作为抽象表示，叫作 DStream。DStream 是随时间推移而收到的数据的序列。在内部，每个时间区间收到的数据都作为 RDD 存在，而 DStream 是由这些 RDD 所组成的序列（因此得名“离散化”）。DStream 可以从各种输入源创建，比如 Flume、Kafka 或者 HDFS。创建出来的 DStream 支持两种操作，一种是转化操作（transformation），会生成一个新的 DStream，另一种是输出操作（output operation），可以把数据写入外部系统中。DStream 提供了许多与 RDD 所支持的操作相类似的操作支持，还增加了与时间相关的新操作，比如滑动窗口。

和批处理程序不同，Spark Streaming 应用需要进行额外配置来保证 24/7 不间断工作。本章会讨论检查点（checkpointing）机制，也就是把数据存储到可靠文件系统（比如 HDFS）上的机制，这也是 Spark Streaming 用来实现不间断工作的主要方式。此外，还会讲到在遇到失败时如何重启应用，以及如何把应用设置为自动重启模式。

最后，就 Spark 1.1 来说，Spark Streaming 只可以在 Java 和 Scala 中使用。试验性的 Python 支持在 Spark 1.2 中引入，不过只支持文本数据。本章就只用 Java 和 Scala 来展示所有的 API，不过类似的概念对 Python 也是适用的。

10.1　一个简单的例子

在开始讲解 Spark Streaming 的细节之前，让我们先来看一个简单的例子。我们会从一台服务器的 7777 端口上收到一个以换行符分隔的多行文本，要从中筛选出包含单词 error 的行，并打印出来。

Spark Streaming 程序最好以使用 Maven 或者 sbt 编译出来的独立应用的形式运行。Spark Streaming 虽然是 Spark 的一部分，它在

Maven 中也以独立工件的形式提供，你也需要在工程中添加一些额外的 import 声明，如例 10-1 至例 10-3 所示。

例 **10-1**：Spark Streaming 的 Maven 索引

```
groupId = org.apache.spark
artifactId = spark-streaming_2.10
version = 1.2.0
```

例 **10-2**：Scala 流计算 import 声明

```
import org.apache.spark.streaming.StreamingContext
import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.dstream.DStream
import org.apache.spark.streaming.Duration
import org.apache.spark.streaming.Seconds
```

例 **10-3**：Java 流计算 import 声明

```
import org.apache.spark.streaming.api.java.JavaStreamingContext;
import org.apache.spark.streaming.api.java.JavaDStream;
import org.apache.spark.streaming.api.java.JavaPairDStream;
import org.apache.spark.streaming.Duration;
import org.apache.spark.streaming.Durations;
```

让我们从创建 StreamingContext 开始，它是流计算功能的主要入口。StreamingContext 会在底层创建出 SparkContext，用来处理数据。其构造函数还接收用来指定多长时间处理一次新数据的批次间隔（batch interval）作为输入，这里我们把它设为 1 秒。接着，调用 `socketTextStream()` 来创建出基于本地 7777 端口上收到的文本数据的 DStream。然后把 DStream 通过 `filter()` 进行转化，只得到包含“error”的行。最后，使用输出操作 `print()` 把一些筛选出来的行打印出来。（如例 10-4 和例 10-5 所示。）

例 **10-4**：用 Scala 进行流式筛选，打印出包含“error”的行

```
// 从SparkConf创建StreamingContext并指定1秒钟的批处理大小
val ssc = new StreamingContext(conf, Seconds(1))
// 连接到本地机器7777端口上后，使用收到的数据创建DStream
val lines = ssc.socketTextStream("localhost", 7777)
// 从DStream中筛选出包含字符串"error"的行
```

```
val errorLines = lines.filter(_.contains("error"))
// 打印出有"error"的行
errorLines.print()
```

例 **10-5**：用 Java 进行流式筛选，打印出包含“error”的行

```
// 从SparkConf创建StreamingContext并指定1秒钟的批处理大小
JavaStreamingContext jssc = new JavaStreamingContext(conf, Durations.secon
// 以端口7777作为输入来源创建DStream
JavaDStream<String> lines = jssc.socketTextStream("localhost", 7777);
// 从DStream中筛选出包含字符串"error"的行
JavaDStream<String> errorLines = lines.filter(new Function<String, Boolean
  public Boolean call(String line) {
    return line.contains("error");
  }});
// 打印出有"error"的行
errorLines.print();
```

这只是设定好了要进行的计算，系统收到数据时计算就会开始。要开始接收数据，必须显式调用 StreamingContext 的 `start()` 方法。这样，Spark Streaming 就会开始把 Spark 作业不断交给下面的 SparkContext 去调度执行。执行会在另一个线程中进行，所以需要调用 `awaitTermination` 来等待流计算完成，来防止应用退出。（见例 10-6 和例 10-7。）

例 **10-6**：用 Scala 进行流式筛选，打印出包含“error”的行

```
// 启动流计算环境StreamingContext并等待它"完成"
ssc.start()
// 等待作业完成
ssc.awaitTermination()
```

例 **10-7**：用 Java 进行流式筛选，打印出包含“error”的行

```
// 启动流计算环境StreamingContext并等待它"完成"
jssc.start();
// 等待作业完成
jssc.awaitTermination();
```

请注意，一个 Streaming context 只能启动一次，所以只有在配置好所有 DStream 以及所需要的输出操作之后才能启动。

现在我们有了一个简易的流计算应用，让我们来运行它，如例 10-8 所示。

例 **10-8**：在 Linux/Mac 操作系统上运行流计算应用并提供数据

```
$ spark-submit --class com.oreilly.learningsparkexamples.scala.StreamingLo
$ASSEMBLY_JAR local[4]

$ nc localhost 7777 # 使你可以键入输入的行来发送给服务器
<此处是你的输入>
```

Windows 用户可以使用 `ncat`（http://nmap.org/ncat/）命令来替代这里的 `nc` 命令。`ncat` 是 `nmap`（http://nmap.org/）工具的一部分。

接下来我们会把这个例子加以扩展以处理 Apache 日志文件。如果你需要生成一些假的日志，可以运行本书 Git 仓库中的脚本 `./bin/fakelogs.sh` 或者 `./bin/fakelogs.cmd` 来把日志发给 7777 端口。

10.2　架构与抽象

Spark Streaming 使用“微批次”的架构，把流式计算当作一系列连续的小规模批处理来对待。Spark Streaming 从各种输入源中读取数据，并把数据分组为小的批次。新的批次按均匀的时间间隔创建出来。在每个时间区间开始的时候，一个新的批次就创建出来，在该区间内收到的数据都会被添加到这个批次中。在时间区间结束时，批次停止增长。时间区间的大小是由批次间隔这个参数决定的。批次间隔一般设在 500 毫秒到几秒之间，由应用开发者配置。每个输入批次都形成一个 RDD，以 Spark 作业的方式处理并生成其他的 RDD。处理的结果可以以批处理的方式传给外部系统。高层次的架构如图 10-1 所示。

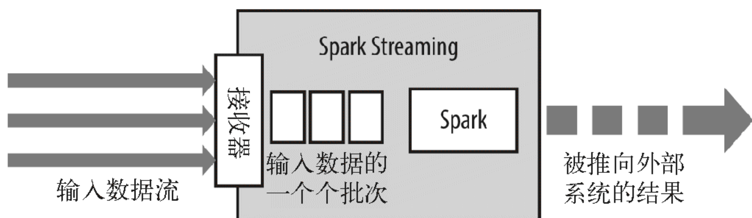


图 10-1：**Spark Streaming** 的高层次架构

我们已经讲到过，Spark Streaming 的编程抽象是离散化流，也就是 DStream（如图 10-2 所示）。它是一个 RDD 序列，每个 RDD 代表数据流中一个时间片内的数据。

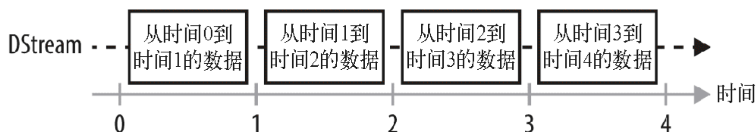


图 10-2：**DStream** 是一个持续的 **RDD** 序列

你可以从外部输入源创建 DStream，也可以对其他 DStream 应用进行转化操作得到新的 DStream。DStream 支持许多第 3 章中所讲到的 RDD 支持的转化操作。另外，DStream 还有"有状态"的转化操作，可以用来聚合不同时间片内的数据。我们会在后面几节进一步讲解。

在列举的简单的例子中，我们以从套接字中收到的数据创建出 DStream，然后对其应用 `filter()` 转化操作。这会在内部创建出如图 10-3 所示的 RDD。

如果运行例 10-8，你会看到与例 10-9 所示近似的输出。

例 **10-9**：运行例 10-8 的日志输出

```
-------------------------------------------
Time: 1413833674000 ms
-------------------------------------------
71.19.157.174 - - [24/Sep/2014:22:26:12 +0000] "GET /error78978 HTTP/1.1"
...
-------------------------------------------
Time: 1413833675000 ms
-------------------------------------------
71.19.164.174 - - [24/Sep/2014:22:27:10 +0000] "GET /error78978 HTTP/1.1"
...
```

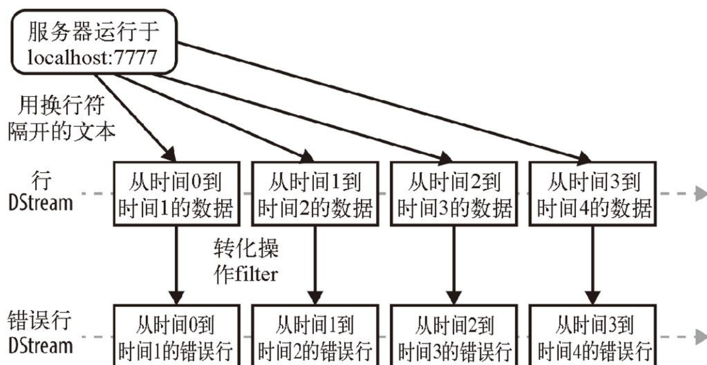


图 10-3：例 10-4 至例 10-8 中的 DStream 及其转化关系

这样的输出清晰地展示了 Spark Streaming 的微批次架构。我们可以看到筛选过的日志每秒钟都被打印一次，这是因为我们创建 StreamingContext 时设定的批次间隔为 1 秒。Spark 用户界面也显示 Spark Streaming 执行了许多小规模作业，如图 10-4 所示。

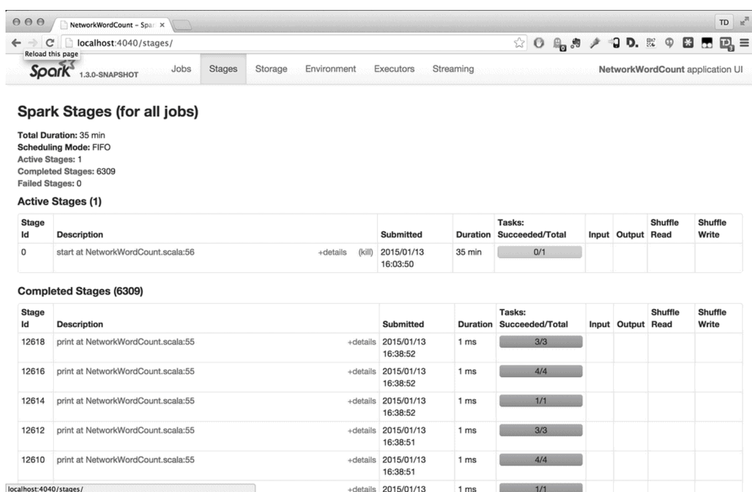


图 10-4：运行流计算作业时的 Spark 应用用户界面

除了转化操作以外，DStream 还支持输出操作，比如在示例中使用的 `print()`。输出操作和 RDD 的行动操作的概念类似。Spark 在行动操作中将数据写入外部系统中，而 Spark Streaming 的输出操作在每个时间区间中周期性执行，每个批次都生成输出。

Spark Streaming 在 Spark 的驱动器程序—工作节点的结构的执行过

程如图 10-5 所示（参照本书前面在图 2-3 中对 Spark 组成部分的描述）。Spark Streaming 为每个输入源启动对应的接收器。接收器以任务的形式运行在应用的执行器进程中，从输入源收集数据并保存为 RDD。它们收集到输入数据后会把数据复制到另一个执行器进程来保障容错性（默认行为）。数据保存在执行器进程的内存中，和缓存 RDD 的方式一样 1。驱动器程序中的 StreamingContext 会周期性地运行 Spark 作业来处理这些数据，把数据与之前时间区间中的 RDD 进行整合。

1在 Spark 1.2 中，接收器也可以把数据备份到 HDFS 上。而且，对于一些输入源，比如 HDFS，天生就是多份存储，所以 Spark Streaming 不会再作一次备份。

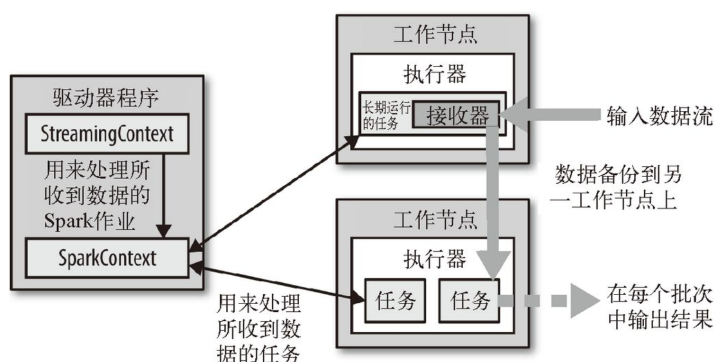


图 **10-5**：**Spark Streaming** 在 **Spark** 各组件中的执行过程

Spark Streaming 对 DStream 提供的容错性与 Spark 为 RDD 所提供的容错性一致：只要输入数据还在，它就可以使用 RDD 谱系重算出任意状态（比如重新执行处理输入数据的操作）。默认情况下，收到的数据分别存在于两个节点上，这样 Spark 可以容忍一个工作节点的故障。不过，如果只用谱系图来恢复的话，重算有可能会花很长时间，因为需要处理从程序启动以来的所有数据。因此，Spark Streaming 也提供了检查点机制，可以把状态阶段性地存储到可靠文件系统中（例如 HDFS 或者 S3）。一般来说，你需要每处理 5-10 个批次的数据就保存一次。在恢复数据时，Spark Streaming 只需要回溯到上一个检查点即可。

接下来会详细探究 Spark Streaming 中的转化操作、输出操作，以及各种输入源。然后，我们会回到容错性和检查点机制，讨论如何为 24/7 不间断运行配置程序。

10.3　转化操作

DStream 的转化操作可以分为无状态（stateless）和有状态（stateful）两种。

- 在无状态转化操作中，每个批次的处理不依赖于之前批次的数据。第 3 章和第 4 章中所讲的常见的 RDD 转化操作，例如 `map()`、`filter()`、`reduceByKey()` 等，都是无状态转化操作。
- 相对地，有状态转化操作需要使用之前批次的数据或者是中间结果来计算当前批次的数据。有状态转化操作包括基于滑动窗口的转化操作和追踪状态变化的转化操作。

10.3.1　无状态转化操作

无状态转化操作就是把简单的 RDD 转化操作应用到每个批次上，也就是转化 DStream 中的每一个 RDD。部分无状态转化操作列在了表 10-1 中。我们已经在图 10-3 中看到了 `filter()` 的例子。第 3 章和 第 4 章讨论的 RDD 转化操作中有不少都可以用于 DStream。注意，针对键值对的 DStream 转化操作（比如 `reduceByKey()`）要添加 `import StreamingContext._` 才能在 Scala 中使用。和 RDD 一样，在 Java 中需要通过 `mapToPair()` 创建出一个 `JavaPairDStream` 才能使用。

表10-1：DStream无状态转化操作的例子（不完整列表）

	用来操作 DStream[T] 的自定义函数的函数签名
对 DStream 中的每个元素应用给定函数，返回由各元素输出的元素组成的 DStream。	[illegible]
对 DStream 中的每个元素应用给定函数，返回由各元素输出的迭代器组成的 DStream。	[illegible]
返回由给定 DStream 中通过筛选的元素组成的 DStream。	[illegible]
改变 DStream 的分区数。	[illegible]
将每个批次中键相同的记录归约。	[illegible]
将每个批次中的记录根据键分组。	[illegible]

需要记住的是，尽管这些函数看起来像作用在整个流上一样，但事实上每个 DStream 在内部是由许多 RDD（批次）组成，且无状态转化操作是分别应用到每个 RDD 上的。例如，`reduceByKey()` 会归约每个时间区间中的数据，但不会归约不同区间之间的数据。我们稍后会讲的有状态转化操作则会整合不同时间区间内的数据。

举个例子，在之前的日志处理程序中，我们可以使用 `map()` 和 `reduceByKey()` 在每个时间区间中对日志根据 IP 地址进行计数，如例 10-10 和例 10-11 所示。

例 **10-10**：在 Scala 中对 DStream 使用 `map()` 和 `reduceByKey()`

```
// 假设ApacheAccessingLog是用来从Apache日志中解析条目的工具类
val accessLogDStream = logData.map(line => ApacheAccessLog.parseFromLogLin
val ipDStream = accessLogsDStream.map(entry => (entry.getIpAddress(), 1))
val ipCountsDStream = ipDStream.reduceByKey((x, y) => x + y)
```

例 **10-11**：在 Java 中对 DStream 使用 `map()` 和 `reduceByKey()`

```
// 假设ApacheAccessingLog是用来从Apache日志中解析条目的工具类
static final class IpTuple implements PairFunction<ApacheAccessLog, String
  public Tuple2<String, Long> call(ApacheAccessLog log) {
    return new Tuple2<>(log.getIpAddress(), 1L);
  }
}

JavaDStream<ApacheAccessLog> accessLogsDStream =
  logData.map(new ParseFromLogLine());
JavaPairDStream<String, Long> ipDStream =
  accessLogsDStream.mapToPair(new IpTuple());
JavaPairDStream<String, Long> ipCountsDStream =
  ipDStream.reduceByKey(new LongSumReducer());
```

无状态转化操作也能在多个 DStream 间整合数据，不过也是在各个时间区间内。例如，键值对 DStream 拥有和 RDD 一样的与连接相关的转化操作，也就是 `cogroup()`、`join()`、`leftOuterJoin()` 等（见 4.3.3 节）。我们可以在 DStream 上使用这些操作，这样就对每个批次分别执行了对应的 RDD 操作。

让我们来看看对两个 DStream 使用连接的一个具体例子。在例 10-12 和例 10-13 中，我们以 IP 地址为键，把请求计数的数据和传输数据量的数据连接起来。

例 **10-12**：在 Scala 中连接两个 DStream

```
val ipBytesDStream =
  accessLogsDStream.map(entry => (entry.getIpAddress(), entry.getContentSi
```

```
val ipBytesSumDStream =
  ipBytesDStream.reduceByKey((x, y) => x + y)
val ipBytesRequestCountDStream =
  ipCountsDStream.join(ipBytesSumDStream)
```

例 **10-13**：在 Java 中连接两个 DStream

```
JavaPairDStream<String, Long> ipBytesDStream =
  accessLogsDStream.mapToPair(new IpContentTuple());
JavaPairDStream<String, Long> ipBytesSumDStream =
  ipBytesDStream.reduceByKey(new LongSumReducer());
JavaPairDStream<String, Tuple2<Long, Long>> ipBytesRequestCountDStream =
  ipCountsDStream.join(ipBytesSumDStream);
```

我们还可以像在常规的 Spark 中一样使用 DStream 的 `union()` 操作将它和另一个 DStream 的内容合并起来，也可以使用 `StreamingContext.union()` 来合并多个流。

最后，如果这些无状态转化操作不够用，DStream 还提供了一个叫作 `transform()` 的高级操作符，可以让你直接操作其内部的 RDD。这个 `transform()` 操作允许你对 DStream 提供任意一个 RDD 到 RDD 的函数。这个函数会在数据流中的每个批次中被调用，生成一个新的流。`transform()` 的一个常见应用就是重用你为 RDD 写的批处理代码。例如，如果你有一个叫作 `extractOutliers()` 的函数，用来从一个日志记录的 RDD 中提取出异常值的 RDD（可能通过对消息进行一些统计），你就可以在 `transform()` 中重用它，如例 10-14 和例 10-15 所示。

例 **10-14**：在 Scala 中对 DStream 使用 `transform()`

```
val outlierDStream = accessLogsDStream.transform { rdd =>
  extractOutliers(rdd)
}
```

例 **10-15**：在 Java 中对 DStream 使用 `transform()`

```
JavaPairDStream<String, Long> ipRawDStream = accessLogsDStream.transform(
  new Function<JavaRDD<ApacheAccessLog>, JavaRDD<ApacheAccessLog>>() {
    public JavaPairRDD<ApacheAccessLog> call(JavaRDD<ApacheAccessLog> rdd)
      return extractOutliers(rdd);
```

```
    }
});
```

你也可以通过 `StreamingContext.transform` 或 `DStream.transformWith(otherStream, func)` 来整合与转化多个 DStream。

10.3.2　有状态转化操作

DStream 的有状态转化操作是跨时间区间跟踪数据的操作；也就是说，一些先前批次的数据也被用来在新的批次中计算结果。主要的两种类型是滑动窗口和 `updateStateByKey()`，前者以一个时间阶段为滑动窗口进行操作，后者则用来跟踪每个键的状态变化（例如构建一个代表用户会话的对象）。

有状态转化操作需要在你的 StreamingContext 中打开检查点机制来确保容错性。我们会在 10.6 节中更详细地讨论检查点机制，现在你只需要知道可以通过传递一个目录作为参数给 `ssc.checkpoint()` 来打开它，如例 10-16 所示。

例 **10-16**：设置检查点

```
ssc.checkpoint("hdfs://...")
```

进行本地开发时，你也可以使用本地路径（例如 /tmp）取代 HDFS。

基于窗口的转化操作

基于窗口的操作会在一个比 StreamingContext 的批次间隔更长的时间范围内，通过整合多个批次的结果，计算出整个窗口的结果。本节会展示如何使用这种转化操作来跟踪网络服务器访问日志中的一些信息，比如常见的一些响应代码、内容大小，以及客户端类型。

所有基于窗口的操作都需要两个参数，分别为窗口时长以及滑动步长，两者都必须是 StreamContext 的批次间隔的整数倍。窗口时长控制每次计算最近的多少个批次的数据，其实就是最近的 `windowDuration/batchInterval` 个批次。如果有一个以 10 秒为批次间隔的源 DStream，要创建一个最近 30 秒的时间窗口（即最近 3 个批次），就应当把 `windowDuration` 设为 30 秒。而滑动步长的默认值与批次间隔相等，用来控制对新的 DStream 进行计算的

间隔。如果源 DStream 批次间隔为 10 秒，并且我们只希望每两个批次计算一次窗口结果，就应该把滑动步长设置为 20 秒。图 10-6 展示了一个例子。

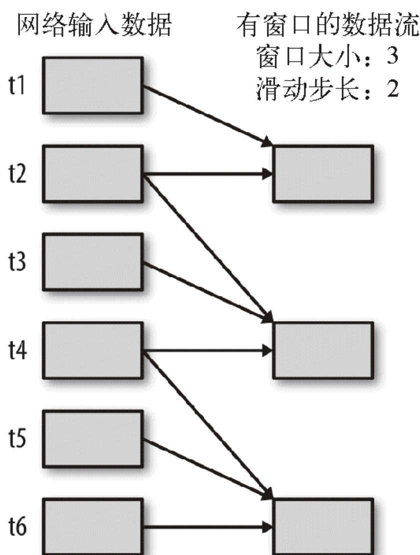


图 10-6：一个基于窗口的流数据，窗口时长为 **3** 个批次，滑动步长为 **2** 个批次；每隔 **2** 个批次就对

对 DStream 可以用的最简单窗口操作是 `window()`，它返回一个新的 DStream 来表示所请求的窗口操作的结果数据。换句话说，`window()` 生成的 DStream 中的每个 RDD 会包含多个批次中的数据，可以对这些数据进行 `count()`、`transform()` 等操作（见例 10-17 和例 10-18）。

例 10-17：如何在 Scala 中使用 `window()` 对窗口进行计数

```
val accessLogsWindow = accessLogsDStream.window(Seconds(30), Seconds(10))
val windowCounts = accessLogsWindow.count()
```

例 10-18：如何在 Java 中使用 `window()` 对窗口进行计数

```
JavaDStream<ApacheAccessLog> accessLogsWindow = accessLogsDStream.window(
    Durations.seconds(30), Durations.seconds(10));
```

```
JavaDStream<Integer> windowCounts = accessLogsWindow.count();
```

尽管可以使用 `window()` 写出所有的窗口操作，Spark Streaming 还是提供了一些其他的窗口操作，让用户可以高效而方便地使用。首先，`reduceByWindow()` 和 `reduceByKeyAndWindow()` 让我们可以对每个窗口更高效地进行归约操作。它们接收一个归约函数，在整个窗口上执行，比如 `+`。除此以外，它们还有一种特殊形式，通过只考虑新进入窗口的数据和离开窗口的数据，让 Spark 增量计算归约结果。这种特殊形式需要提供归约函数的一个逆函数，比如 `+` 对应的逆函数为 `-`。对于较大的窗口，提供逆函数可以大大提高执行效率（见图 10-7）。

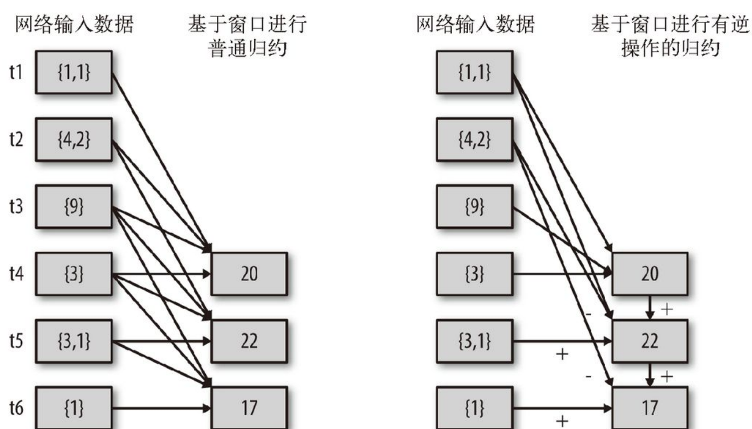


图 **10-7**：普通的 **`reduceByWindow()`** 与使用逆函数的增量式 **`reduceByWindow()`** 的区别

在日志处理的例子中，我们可以使用这两个函数来更高效地对每个 IP 地址访问量进行计数，如例 10-19 和例 10-20 所示。

例 **10-19**：Scala 版本的每个 IP 地址的访问量计数

```
val ipDStream = accessLogsDStream.map(logEntry => (logEntry.getIpAddress()
val ipCountDStream = ipDStream.reduceByKeyAndWindow(
  {(x, y) => x + y}, // 加上新进入窗口的批次中的元素
  {(x, y) => x - y}, // 移除离开窗口的老批次中的元素
  Seconds(30),       // 窗口时长
  Seconds(10))       // 滑动步长
```

例 **10-20**：Java 版本的每个 IP 地址的访问量计数

```
class ExtractIp extends PairFunction<ApacheAccessLog, String, Long> {
  public Tuple2<String, Long> call(ApacheAccessLog entry) {
    return new Tuple2(entry.getIpAddress(), 1L);
  }
}
class AddLongs extends Function2<Long, Long, Long>() {
  public Long call(Long v1, Long v2) { return v1 + v2; }
}
class SubtractLongs extends Function2<Long, Long, Long>() {
  public Long call(Long v1, Long v2) { return v1 - v2; }
}

JavaPairDStream<String, Long> ipAddressPairDStream = accessLogsDStream.map
  new ExtractIp());

JavaPairDStream<String, Long> ipCountDStream = ipAddressPairDStream.
  reduceByKeyAndWindow(
  new AddLongs(), // 加上新进入窗口的批次中的元素
  new SubtractLongs()
  // 移除离开窗口的老批次中的元素
  Durations.seconds(30),  // 窗口时长
  Durations.seconds(10)); // 滑动步长
```

最后，DStream 还提供了 `countByWindow()` 和 `countByValueAndWindow()` 作为对数据进行计数操作的简写。`countByWindow()` 返回一个表示每个窗口中元素个数的 `DStream`，而 `countByValueAndWindow()` 返回的 DStream 则包含窗口中每个值的个数，如例 10-21 和例 10-22 所示。

例 **10-21**：Scala 中的窗口计数操作

```
val ipDStream = accessLogsDStream.map{entry => entry.getIpAddress()}
val ipAddressRequestCount = ipDStream.countByValueAndWindow(Seconds(30), S
val requestCount = accessLogsDStream.countByWindow(Seconds(30), Seconds(10
```

例 **10-22**：Java 中的窗口计数操作

```
JavaDStream<String> ip = accessLogsDStream.map(
  new Function<ApacheAccessLog, String>() {
    public String call(ApacheAccessLog entry) {
      return entry.getIpAddress();
    }});
JavaDStream<Long> requestCount = accessLogsDStream.countByWindow(
  Dirations.seconds(30), Durations.seconds(10));
```

```
JavaPairDStream<String, Long> ipAddressRequestCount = ip.countByValueAndWi
  Dirations.seconds(30), Durations.seconds(10));
```

UpdateStateByKey转化操作

有时，我们需要在 DStream 中跨批次维护状态（例如跟踪用户访问网站的会话）。针对这种情况，`updateStateByKey()` 为我们提供了对一个状态变量的访问，用于键值对形式的 DStream。给定一个由（键，事件）对构成的 DStream，并传递一个指定如何根据新的事件更新每个键对应状态的函数，它可以构建出一个新的 DStream，其内部数据为（键，状态）对。例如，在网络服务器日志中，事件可能是对网站的访问，此时键是用户的 ID。使用 `updateStateByKey()` 可以跟踪每个用户最近访问的 10 个页面。这个列表就是“状态”对象，我们会在每个事件到来时更新这个状态。

要使用 `updateStateByKey()`，提供了一个 `update(events, oldState)` 函数，接收与某键相关的事件以及该键之前对应的状态，返回这个键对应的新状态。这个函数的签名如下所示。

- `events`：是在当前批次中收到的事件的列表（可能为空）。
- `oldState`：是一个可选的状态对象，存放在 `Option` 内；如果一个键没有之前的状态，这个值可以空缺。
- `newState`：由函数返回，也以 `Option` 形式存在；我们可以返回一个空的 `Option` 来表示想要删除该状态。

`updateStateByKey()` 的结果会是一个新的 DStream，其内部的 RDD 序列是由每个时间区间对应的（键，状态）对组成的。

举个简单的例子，使用 `updateStateByKey()` 来跟踪日志消息中各 HTTP 响应代码的计数。这里的键是响应代码，状态是代表各响应代码计数的整数，事件则是页面访问。请注意，跟之前的窗口例子不同的是，例 10-23 和例 10-24 会进行自程序启动开始就“无限增长”的计数。

例 **10-23**：在 Scala 中使用 `updateStateByKey()` 运行响应代码的计数

```
def updateRunningSum(values: Seq[Long], state: Option[Long]) = {
  Some(state.getOrElse(0L) + values.size)
}
```

```
val responseCodeDStream = accessLogsDStream.map(log => (log.getResponseCod
val responseCodeCountDStream = responseCodeDStream.updateStateByKey(update
```

例 **10-24**：在 Java 中使用 `updateStateByKey()` 运行响应代码的计数

```
class UpdateRunningSum implements Function2<List<Long>,
    Optional<Long>, Optional<Long>> {
  public Optional<Long> call(List<Long> nums, Optional<Long> current) {
    long sum = current.or(0L);
    return Optional.of(sum + nums.size());
  }
};

JavaPairDStream<Integer, Long> responseCodeCountDStream = accessLogsDStrea
    new PairFunction<ApacheAccessLog, Integer, Long>() {
      public Tuple2<Integer, Long> call(ApacheAccessLog log) {
        return new Tuple2(log.getResponseCode(), 1L);
    }})
  .updateStateByKey(new UpdateRunningSum());
```

10.4　输出操作

输出操作指定了对流数据经转化操作得到的数据所要执行的操作（例如把结果推入外部数据库或输出到屏幕上）。



与 RDD 中的惰性求值类似，如果一个 DStream 及其派生出的 DStream 都没有被执行输出操作，那么这些 DStream 就都不会被求值。如果 StreamingContext 中没有设定输出操作，整个 context 就都不会启动。

常用的一种调试性输出操作是 `print()`，它会在每个批次中抓取 DStream 的前十个元素打印出来。

一旦调试好了程序，就可以使用输出操作来保存结果了。Spark Streaming 对于 DStream 有与 Spark 类似的 `save()` 操作，它们接受一个目录作为参数来存储文件，还支持通过可选参数来设置文件的后缀名。每个批次的结果被保存在给定目录的子目录中，且文件名中含有时间和后缀名。我们可以把 IP 地址计数保存起来，如例 10-25 所示。

例 **10-25**：在 Scala 中将 DStream 保存为文本文件

```
ipAddressRequestCount.saveAsTextFiles("outputDir", "txt")
```

还有一个更为通用的 `saveAsHadoopFiles()` 函数，接收一个 Hadoop 输出格式作为参数。例如，Spark Streaming 没有内建的 `saveAsSequenceFile()` 函数，但是我们可以使用例 10-26 和例 10-27 中的方法来保存 SequenceFile 文件。

例 **10-26**：在 Scala 中将 DStream 保存为 SequenceFile

```
val writableIpAddressRequestCount = ipAddressRequestCount.map {
  (ip, count) => (new Text(ip), new LongWritable(count)) }
writableIpAddressRequestCount.saveAsHadoopFiles[
  SequenceFileOutputFormat[Text, LongWritable]]("outputDir", "txt")
```

例 **10-27** ：在 Java 中将 DStream 保存为 SequenceFile

```
JavaPairDStream<Text, LongWritable> writableDStream = ipDStream.mapToPair(
  new PairFunction<Tuple2<String, Long>, Text, LongWritable>() {
    public Tuple2<Text, LongWritable> call(Tuple2<String, Long> e) {
      return new Tuple2(new Text(e._1()), new LongWritable(e._2()));
  }});
class OutFormat extends SequenceFileOutputFormat<Text, LongWritable> {};
writableDStream.saveAsHadoopFiles(
  "outputDir", "txt", Text.class, LongWritable.class, OutFormat.class);
```

最后，还有一个通用的输出操作 `foreachRDD()`，它用来对 DStream 中的 RDD 运行任意计算。这和 `transform()` 有些类似，都可以让我们访问任意 RDD。在 `foreachRDD()` 中，可以重用我们在 Spark 中实现的所有行动操作。比如，常见的用例之一是把数据写到诸如 MySQL 的外部数据库中。对于这种操作，Spark 没有提供对应的 `saveAs()` 函数，但可以使用 RDD 的 `eachPartition()` 方法来把它写出去。为了方便，`foreachRDD()` 也可以提供给我们当前批次的时间，允许我们把不同时间的输出结果存到不同的位置。参见例 10-28。

例 **10-28**：在 Scala 中使用 `foreachRDD()` 将数据存储到外部系统中

```
ipAddressRequestCount.foreachRDD { rdd =>
  rdd.foreachPartition { partition =>
    // 打开到存储系统的连接（比如一个数据库的连接）
    partition.foreach { item =>
      // 使用连接把item存到系统中
    }
    // 关闭连接
  }
}
```

10.5　输入源

Spark Streaming 原生支持一些不同的数据源。一些“核心”数据源已经被打包到 Spark Streaming 的 Maven 工件中，而其他的一些则可以通过 `spark-streaming-kafka` 等附加工件获取。

本节会对部分数据源进行一些概述。我们假定你已经安装好了这些数据源，并且不会介绍这些系统中与 Spark 无关的组件。如果你在设计一个新的应用，我们建议你从使用 HDFS 或 Kafka 这种简单的输入源开始。

10.5.1　核心数据源

所有用来从核心数据源创建 DStream 的方法都位于 StreamingContext 中。我们已经在示例中用过其中一个：套接字。这里再讨论两个：文件和 Akka actor。

1. 文件流

因为 Spark 支持从任意 Hadoop 兼容的文件系统中读取数据，所以 Spark Streaming 也就支持从任意 Hadoop 兼容的文件系统目录中的文件创建数据流。由于支持多种后端，这种方式广为使用，尤其是对于像日志这样始终要复制到 HDFS 上的数据。要让 Spark Streaming 来处理数据，我们需要为目录名字提供统一的日期格式，文件也必须原子化创建（比如把文件移入 Spark 监控的目录）。2 可以修改例 10-4 和例 10-5 来处理新出现在一个目录下的日志文件，如例 10-29 和例 10-30 所示。

> 2原子化表示整个操作一次完成。如果 Spark Streaming 将要处理文件时，更多的数据出现了，Spark Streaming 就会无法注意到新添加的数据，因此原子化在这里很重要。在文件系统中，文件重命名操作一般是原子化的。

例 10-29：用 Scala 读取目录中的文本文件流

```
val logData = ssc.textFileStream(logDirectory)
```

例 10-30：用 Java 读取目录中的文本文件流

```
JavaDStream<String> logData = jssc.textFileStream(logsDirectory);
```

我们可以使用所提供的 ./bin/fakelogs_directory.sh 脚本来造出假日志。如果有真实日志数据的话，也可以用 `mv` 命令将日志文件循环移入所监控的目录中。

除了文本数据，也可以读入任意 Hadoop 输入格式。与 5.2.6 节所讲的一样，只需要将 `Key`、`Value` 以及 `InputFormat` 类提供给 Spark Streaming 即可。例如，如果先前已经有了一个流处理作业来处理日志，并已经将得到的每个时间区间内传输的数据分别存储成了一个 SequenceFile，就可以如例 10-31 中所示的那样来读取数据。

例 10-31：用 Scala 读取目录中的 SequenceFile 流

```
ssc.fileStream[LongWritable, IntWritable,
    SequenceFileInputFormat[LongWritable, IntWritable]](inputDirectory).ma
    case (x, y) => (x.get(), y.get())
}
```

2. Akka actor流

另一个核心数据源接收器是 `actorStream`，它可以把 Akka actor（http://akka.io/）作为数据流的源。要创建出一个 actor 流，需要创建一个 Akka actor，然后实现 `org.apache.spark.streaming.receiver.ActorHelper` 接口。要把输入数据从 actor 复制到 Spark Streaming 中，需要在收到新数据时调用 actor 的 `store()` 函数。Akka actor 流不是很常见，所以我们不会对其进行深入探究。你可以阅读流计算的文档（http://spark.apache.org/docs/latest/streaming-custom-receivers.html）以及 Spark 中的ActorWordCount（https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/streaming/ActorWordCount.scala）的例子来学习如何使用它们。

10.5.2　附加数据源

除核心数据源外，还可以用附加数据源接收器来从一些知名数据获取系统中接收的数据，这些接收器都作为 Spark Streaming 的组件进行独立打包了。它们仍然是 Spark 的一部分，不过你需要在构建文件中添加额外的包才能使用它们。现有的接收器包括 Twitter、Apache Kafka、Amazon Kinesis、Apache Flume，以及 ZeroMQ。可以通过添加与 Spark 版本匹配的 Maven 工件 `spark-streaming-[projectname]_2.10` 来引入这些附加接收器。

1. Apache Kafka

Apache Kafka（http://kafka.apache.org/）因其速度与弹性成为了一个流行的输入源。使用 Kafka 原生的支持，可以轻松处理许多主题的消息。在工程中需要引入 Maven 工件 `spark-streaming-kafka_2.10` 来使用它。包内提供的 `KafkaUtils` 对象可以在 StreamingContext 和 JavaStreamingContext 中以你的 Kafka 消息创建出 DStream。由于 `KafkaUtils` 可以订阅多个主题，因此它创建出的 DStream 由成对的主题和消息组成。要创建出一个流数据，需要使用 StreamingContext 实例、一个由逗号隔开的 ZooKeeper 主机列表字符串、消费者组的名字（唯一名字），以及一个从主题到针对这个主题的接收器线程数的映射表来调用 `createStream()` 方法（如例 10-32 和例 10-33 所示）。

例 10-32：在 Scala 中用 Apache Kafka 订阅 Panda 主题

```
import org.apache.spark.streaming.kafka._
...
// 创建一个从主题到接收器线程数的映射表
val topics = List(("pandas", 1), ("logs", 1)).toMap
val topicLines = KafkaUtils.createStream(ssc, zkQuorum, group, topics)
StreamingLogInput.processLines(topicLines.map(_._2))
```

例 10-33：在 Java 中用 Apache Kafka 订阅 Panda 主题

```
import org.apache.spark.streaming.kafka.*;
...
// 创建一个从主题到接收器线程数的映射表
Map<String, Integer> topics = new HashMap<String, Integer>();
topics.put("pandas", 1);
```

```
topics.put("logs", 1);
JavaPairDStream<String, String> input =
  KafkaUtils.createStream(jssc, zkQuorum, group, topics);
input.print();
```

2. Apache Flume

Spark 提供两个不同的接收器来使用 Apache Flume（http://flume.apache.org/，见图 10-8）。两个接收器简介如下。

- 推式接收器

  该接收器以 Avro 数据池的方式工作，由 Flume 向其中推数据。

- 拉式接收器

  该接收器可以从自定义的中间数据池中拉数据，而其他进程可以使用 Flume 把数据推进该中间数据池。

两种方式都需要重新配置 Flume，并在某个节点配置的端口上运行接收器（不是已有的 Spark 或者 Flume 使用的端口）。要使用其中任何一种方法，都需要在工程中引入 Maven 工件 `spark-streaming-flume_2.10`。

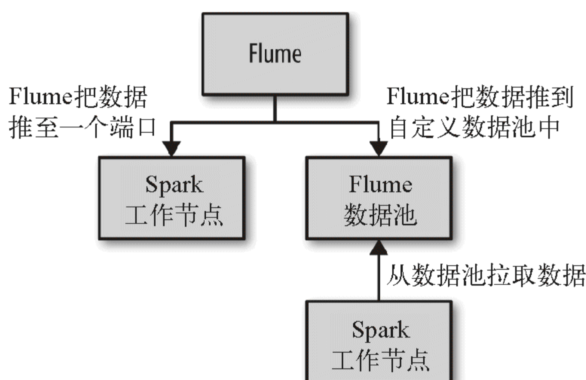


图 10-8：**Flume** 接收器选项

3. 推式接收器

推式接收器的方法设置起来很容易，但是它不使用事务来接收数据。

在这种方式中，接收器以 Avro 数据池的方式工作，我们需要配置 Flume 来把数据发到 Avro 数据池（见例 10-34）。我们提供的 `FlumeUtils` 对象会把接收器配置在一个特定的工作节点的主机名及端口号上（见例 10-35 和例 10-36）。这些设置必须和 Flume 配置相匹配。

例 **10-34**：Flume 对 Avro 池的配置

```
a1.sinks = avroSink
a1.sinks.avroSink.type = avro
a1.sinks.avroSink.channel = memoryChannel
a1.sinks.avroSink.hostname = receiver-hostname
a1.sinks.avroSink.port = port-used-for-avro-sink-not-spark-port
```

例 **10-35**：Scala 中的 FlumeUtils 代理

```
val events = FlumeUtils.createStream(ssc, receiverHostname, receiverPort)
```

例 **10-36**：Java 中的 FlumeUtils 代理

```
JavaDStream<SparkFlumeEvent> events = FlumeUtils.createStream(ssc, receive
```

虽然这种方式很简洁，但缺点是没有事务支持。这会增加运行接收器的工作节点发生错误时丢失少量数据的几率。不仅如此，如果运行接收器的工作节点发生故障，系统会尝试从另一个位置启动接收器，这时需要重新配置 Flume 才能将数据发给新的工作节点。这样配置会比较麻烦。

4. 拉式接收器

较新的方式是拉式接收器（在 Spark 1.1 中引入），它设置了一个专用的 Flume 数据池供 Spark Streaming 读取，并让接收器主动从数据池中拉取数据。这种方式的优点在于弹性较好，Spark Streaming 通过事务从数据池中读取并复制数据。在收到事务完成的通知前，这些数据还保留在数据池中。

我们需要先把自定义数据池配置为 Flume 的第三方插件。安装插件的最新方法请参考 Flume 文档的相关部分（https://flume.apache.org/FlumeUserGuide.html#installing-third-party-plugins）。由于插件是用 Scala 写的，因此需要把插件本身以及 Scala 库都添加到 Flume 插

件中。Spark 1.1 中对应的 Maven 索引如例 10-37 所示。

例 **10-37**：Flume 数据池的 Maven 索引

```
groupId = org.apache.spark
artifactId = spark-streaming-flume-sink_2.10
version = 1.2.0

groupId = org.scala-lang
artifactId = scala-library
version = 2.10.4
```

当你把自定义 Flume 数据池添加到一个节点上之后，就需要配置 Flume 来把数据推送到这个数据池中，如例 10-38 所示。

例 **10-38**：Flume 对自定义数据池的配置

```
a1.sinks = spark
a1.sinks.spark.type = org.apache.spark.streaming.flume.sink.SparkSink
a1.sinks.spark.hostname = receiver-hostname
a1.sinks.spark.port = port-used-for-sync-not-spark-port
a1.sinks.spark.channel = memoryChannel
```

等到数据已经在数据池中缓存起来，就可以调用 `FlumeUtils` 来读取数据了，如例 10-39 和 10-40 所示。

例 **10-39**：在 Scala 中使用 `FlumeUtils` 读取自定义数据池

```
val events = FlumeUtils.createPollingStream(ssc, receiverHostname, receive
```

例 **10-40**：在 Java 中使用 `FlumeUtils` 读取自定义数据池

```
JavaDStream<SparkFlumeEvent> events = FlumeUtils.createPollingStream(ssc,
  receiverHostname, receiverPort)
```

在这两个例子中，DStream 是由 SparkFlumeEvent（https://spark.apache.org/docs/latest/api/java/org/apache/spark/streaming/flume/SparkFlumeEvent.html）组成的。可以通过

event 访问下层的 AvroFlumeEvent。如果事件主体是 UTF-8 字符串，就可以用例 10-41 所示的方式获取其内容。

例 **10-41**：Scala 中的 SparkFlumeEvent

```
// 假定flume事件是UTF-8编码的日志记录
val lines = events.map{e => new String(e.event.getBody().array(), "UTF-8")
```

5. 自定义输入源

除了上述这些源，你也可以实现自己的接收器来支持别的输入源。详细信息请参考 Spark 文档中的“自定义流计算接收器指南”（Streaming Custom Receivers guide，http://spark.apache.org/docs/latest/streaming-custom-receivers.html）。

10.5.3 多数据源与集群规模

如前文所述，可以使用类似 union() 这样的操作将多个 DStream 合并。通过这些操作符，可以把多个输入的 DStream 合并起来。有时，使用多个接收器对于提高聚合操作中的数据获取的吞吐量非常必要（如果只用一个接收器，可能会成为性能瓶颈）。另外，有时我们需要用不同的接收器来从不同的输入源中接收各种数据，然后使用 join 或 cogroup 进行整合。

理解接收器是如何在 Spark 集群中运行的，对于我们使用多个接收器至关重要。每个接收器都以 Spark 执行器程序中一个长期运行的任务的形式运行，因此会占据分配给应用的 CPU 核心。此外，我们还需要有可用的 CPU 核心来处理数据。这意味着如果要运行多个接收器，就必须至少有和接收器数目相同的核心数，还要加上用来完成计算所需要的核心数。例如，如果我们想要在流计算应用中运行 10 个接收器，那么至少需要为应用分配 11 个 CPU 核心。

 不要在本地模式下把主节点配置为 "local" 或 "local[1]" 来运行 Spark Streaming 程序。这种配置只会分配一个 CPU 核心给任务，如果接收器运行在这样的配置里，就没有剩余的资源来处理收到的数据了。至少要使用 "local[2]" 来利用更多的核心。

10.6　24/7不间断运行

Spark Streaming 的一大优势在于它提供了强大的容错性保障。只要输入数据存储在可靠的系统中，Spark Streaming 就可以根据输入计算出正确的结果，提供“精确一次”执行的语义（就好像所有的数据都是在没有任何节点失败的情况下处理的一样），即使是工作节点或者驱动器程序发生了失败。

要不间断运行 Spark Streaming 应用，需要一些特别的配置。第一步是设置好诸如 HDFS 或 Amazon S3 等可靠存储系统中的检查点机制。3 不仅如此，我们还需要考虑驱动器程序的容错性（需要特别的配置代码）以及对不可靠输入源的处理。本节会介绍如何进行这些配置。

> 3我们不会介绍如何配置这些文件系统，不过它们一般都会在 Hadoop 环境或者云环境中已经配好。如果你在部署自己的集群，配置 HDFS 可能是最容易的。

10.6.1　检查点机制

检查点机制是我们在 Spark Streaming 中用来保障容错性的主要机制。它可以使 Spark Streaming 阶段性地把应用数据存储到诸如 HDFS 或 Amazon S3 这样的可靠存储系统中，以供恢复时使用。具体来说，检查点机制主要为以下两个目的服务。

- 控制发生失败时需要重算的状态数。我们在 10.2 节中讨论过，Spark Streaming 可以通过转化图的谱系图来重算状态，检查点机制则可以控制需要在转化图中回溯多远。
- 提供驱动器程序容错。如果流计算应用中的驱动器程序崩溃了，你可以重启驱动器程序并让驱动器程序从检查点恢复，这样 Spark Streaming 就可以读取之前运行的程序处理数据的进度，并从那里继续。

出于这些原因，检查点机制对于任何生产环境中的流计算应用都至关重要。你可以通过向 `ssc.checkpoint()` 方法传递一个路径参数（HDFS、S3 或者本地路径均可）来配置检查点机制，如例 10-42 所示。

例 **10-42**：配置检查点

```
ssc.checkpoint("hdfs://...")
```

注意，即便是在本地模式下，如果你尝试运行一个有状态操作而没有打开检查点机制，Spark Streaming 也会给出提示。此时，你需要使用一个本地文件系统中的路径来打开检查点。不过，在所有的生产环境配置中，你都应当使用诸如 HDFS、S3 或者网络文件系统这样的带备份的系统。

10.6.2　驱动器程序容错

驱动器程序的容错要求我们以特殊的方式创建 StreamingContext。我们需要把检查点目录提供给 StreamingContext。与直接调用 `new StreamingContext` 不同，应该使用 `StreamingContext.getOrCreate()` 函数。应把之前的示例中的代码改成如例 10-43 和例 10-44 所示的那样。

例 10-43：用 Scala 配置一个可以从错误中恢复的驱动器程序

```
def createStreamingContext() = {
  ...
  val sc = new SparkContext(conf)
  // 以1秒作为批次大小创建StreamingContext
  val ssc = new StreamingContext(sc, Seconds(1))
  ssc.checkpoint(checkpointDir)
}
...
val ssc = StreamingContext.getOrCreate(checkpointDir, createStreamingConte
```

例 10-44：用 Java 配置一个可以从错误中恢复的驱动器程序

```
JavaStreamingContextFactory fact = new JavaStreamingContextFactory() {
  public JavaStreamingContext call() {
    ...
    JavaSparkContext sc = new JavaSparkContext(conf);
    // 以1秒作为批次大小创建StreamingContext
    JavaStreamingContext jssc = new JavaStreamingContext(sc, Durations.sec
    jssc.checkpoint(checkpointDir);
    return jssc;
  }};
JavaStreamingContext jssc = JavaStreamingContext.getOrCreate(checkpointDi
```

当这段代码第一次运行时，假设检查点目录还不存在，那么 StreamingContext 会在你调用工厂函数（在 Scala 中为 `createStreamingContext()`，在 Java 中为 `JavaStreamingContextFactory()`）时把目录创建出来。此处你需要设置检查点目录。在驱动器程序失败之后，如果你重启驱动器程序并再次执行代码，`getOrCreate()` 会重新从检查点目录中初始化出 StreamingContext，然后继续处理。

除了用 `getOrCreate()` 来实现初始化代码以外，你还需要编写在驱动器程序崩溃时重启驱动器进程的代码。在大多数集群管理器中，Spark 不会在驱动器程序崩溃时自动重启驱动器进程，所以你需要使用诸如 `monit` 这样的工具来监视驱动器进程并进行重启。最佳的实现方式往往取决于你的具体环境。Spark 在独立集群管理器中提供了更丰富的支持，可以在提交驱动器程序时使用 `--supervise` 标记来让 Spark 重启失败的驱动器程序。你还要传递 `--deploy-mode cluster` 参数来确保驱动器程序在集群中运行，而不是在本地机器上运行，如例 10-45 所示。

例 **10-45**：使用监管模式启动驱动器程序

```
./bin/spark-submit --deploy-mode cluster --supervise --master spark://...
```

在使用这个选项时，如果你希望 Spark 独立模式集群的主节点也是容错的，就可以通过 ZooKeeper 来配置主节点的容错性，详细信息请参考 Spark 的文档（https://spark.apache.org/docs/latest/spark-standalone.html#high-availability）。这样配置之后，就不用再担心你的应用会出现单个节点失败的情况。

最后需要说明的是，当驱动器程序崩溃时，Spark 的执行器进程也会重启。这有可能会在以后的 Spark 版本中有所改变，不过这是 1.2 以及更早版本的 Spark 的预期的行为，因为执行器程序不能在没有驱动器程序的情况下继续处理数据。重启驱动器程序会启动新的执行器进程来继续之前的计算。

10.6.3　工作节点容错

为了应对工作节点失败的问题，Spark Streaming 使用与 Spark 的容错机制相同的方法。所有从外部数据源中收到的数据都在多个工作节点上备份。所有从备份数据转化操作的过程中创建出来的 RDD 都能容忍一个工作节点的失败，因为根据 RDD 谱系图，系统可以把丢失

的数据从幸存的输入数据备份中重算出来。

10.6.4　接收器容错

运行接收器的工作节点的容错也是很重要的。如果这样的节点发生错误，Spark Streaming 会在集群中别的节点上重启失败的接收器。然而，这种情况会不会导致数据的丢失取决于数据源的行为（数据源是否会重发数据）以及接收器的实现（接收器是否会向数据源确认收到数据）。举个例子，使用 Flume 作为数据源时，两种接收器的主要区别在于数据丢失时的保障。在“接收器从数据池中拉取数据”的模型中，Spark 只会在数据已经在集群中备份时才会从数据池中移除元素。而在“向接收器推数据”的模型中，如果接收器在数据备份之前失败，一些数据可能就会丢失。总的来说，对于任意一个接收器，你必须同时考虑上游数据源的容错性（是否支持事务）来确保零数据丢失。

总的来说，接收器提供以下保证。

- 所有从可靠文件系统中读取的数据（比如通过 `StreamingContext.hadoopFiles` 读取的）都是可靠的，因为底层的文件系统是有备份的。Spark Streaming 会记住哪些数据存放到了检查点中，并在应用崩溃后从检查点处继续执行。

- 对于像 Kafka、推式 Flume、Twitter 这样的不可靠数据源，Spark 会把输入数据复制到其他节点上，但是如果接收器任务崩溃，Spark 还是会丢失数据。在 Spark 1.1 以及更早的版本中，收到的数据只被备份到执行器进程的内存中，所以一旦驱动器程序崩溃（此时所有的执行器进程都会丢失连接），数据也会丢失。在 Spark 1.2 中，收到的数据被记录到诸如 HDFS 这样的可靠的文件系统中，这样即使驱动器程序重启也不会导致数据丢失。

综上所述，确保所有数据都被处理的最佳方式是使用可靠的数据源（例如 HDFS、拉式 Flume 等）。如果你还要在批处理作业中处理这些数据，使用可靠数据源是最佳方式，因为这种方式确保了你的批处理作业和流计算作业能读取到相同的数据，因而可以得到相同的结果。

10.6.5　处理保证

由于 Spark Streaming 工作节点的容错保障，Spark Streaming 可以为所有的转化操作提供“精确一次”执行的语义，即使一个工作节点在处理部分数据时发生失败，最终的转化结果（即转化操作得到的 RDD）仍然与数据只被处理一次得到的结果一样。

然而，当把转化操作得到的结果使用输出操作推入外部系统中时，写结果的任务可能因故障而执行多次，一些数据可能也就被写了多次。由于这引入了外部系统，因此我们需要专门针对各系统的代码来处理这样的情况。我们可以使用事务操作来写入外部系统（即原子化地将一个 RDD 分区一次写入），或者设计幂等的更新操作（即多次运行同一个更新操作仍生成相同的结果）。比如 Spark Streaming 的 saveAs...File 操作会在一个文件写完时自动将其原子化地移动到最终位置上，以此确保每个输出文件只存在一份。

10.7　Streaming用户界面

Spark Streaming 提供了一个特殊的用户界面，可以让我们查看应用在干什么。这个界面在常规的 Spark 用户界面（一般为 http://:4040）上的 Streaming 标签页里。图 10-9 是 Streaming 用户界面的一个截屏。

Streaming 用户界面展示了批处理和接收器的统计信息。在所列举的例子中，有一个网络接收器，借此可以看到消息处理的速率。如果处理速度缓慢，就可以看到每个接收器可以处理多少条数据，也可以看到接收器是否发生了故障。批处理的统计信息则向我们呈现批处理已经占用的时长，以及调度作业时的延迟情况。如果集群遇到了资源竞争，那么调度的延迟会增长。

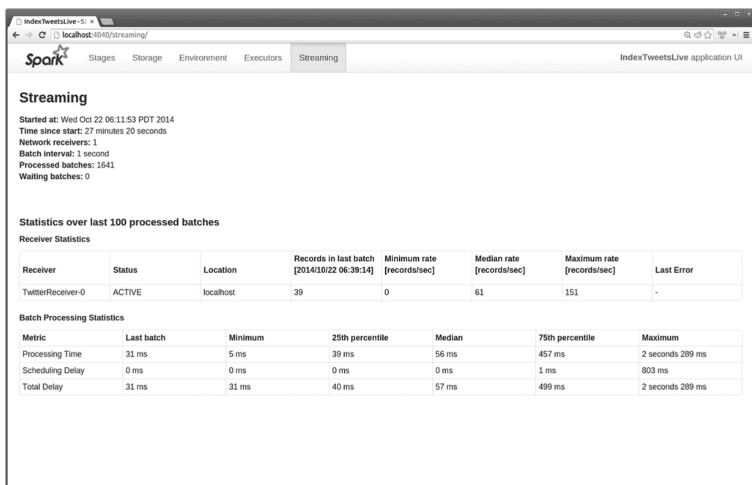


图 10-9：**Spark** 用户界面中的 **Streaming** 用户界面标签页

10.8　性能考量

除了我们已经针对一般的 Spark 应用讨论过的性能考量以外，Spark Streaming 应用还有一些特殊的调优选项。

10.8.1　批次和窗口大小

最常见的问题是 Spark Streaming 可以使用的最小批次间隔是多少。总的来说，500 毫秒已经被证实为对许多应用而言是比较好的最小批次大小。寻找最小批次大小的最佳实践是从一个比较大的批次大小（10 秒左右）开始，不断使用更小的批次大小。如果 Streaming 用户界面中显示的处理时间保持不变，你就可以进一步减小批次大小。如果处理时间开始增加，你可能已经达到了应用的极限。

相似地，对于窗口操作，计算结果的间隔（也就是滑动步长）对于性能也有巨大的影响。当计算代价巨大并成为系统瓶颈时，就应该考虑提高滑动步长了。

10.8.2　并行度

减少批处理所消耗时间的常见方式还有提高并行度。有以下三种方式可以提高并行度。

- 增加接收器数目

  有时如果记录太多导致单台机器来不及读入并分发的话，接收器会成为系统瓶颈。这时你就需要通过创建多个输入 DStream（这样会创建多个接收器）来增加接收器数目，然后使用 `union` 来把数据合并为一个数据源。

- 将收到的数据显式地重新分区

  如果接收器数目无法再增加，你可以通过使用 DStream.repartition 来显式重新分区输入流（或者合并多个流得到的数据流）来重新分配收到的数据。

- 提高聚合计算的并行度

  对于像 `reduceByKey()` 这样的操作，你可以在第二个参数中指定并行度，我们在介绍RDD 时提到过类似的手段。

10.8.3　垃圾回收和内存使用

Java 的垃圾回收机制（简称 GC）也可能会引起问题。你可以通过打开 Java 的并发标志—清除收集器（Concurrent Mark-Sweep garbage collector）来减少 GC 引起的不可预测的长暂停。并发标志—清除收集器总体上会消耗更多的资源，但是会减少暂停的发生。

可以通过在配置参数 `spark.executor.extraJavaOptions` 中添加 `-XX:+UseConcMarkSweepGC` 来控制选择并发标志—清除收集器。例 10-46 展示了使用 `spark-submit` 时的配置方法。

例 **10-46**：打开并发标志—清除收集器

```
spark-submit --conf spark.executor.extraJavaOptions=-XX:+UseConcMarkSweepG
```

除了使用较少引发暂停的垃圾回收器，你还可以通过减轻 GC 的压力来大幅度改善性能。把 RDD 以序列化的格式缓存（而不使用原生的对象）也可以减轻 GC 的压力，这也是为什么默认情况下 Spark Streaming 生成的 RDD 都以序列化后的格式存储。使用 Kryo 序列化工具可以进一步减少缓存在内存中的数据所需要的内存大小。

Spark 也允许我们控制缓存下来的 RDD 以怎样的策略从缓存中移除。默认情况下，Spark 使用 LRU 缓存。如果你设置了

`spark.cleaner.ttl`，Spark 也会显式移除超出给定时间范围的老 RDD。主动从缓存中移除不大可能再用到的 RDD，可以减轻 GC 的压力。

10.9　总结

本章我们学习了如何使用 DStream 操作流数据。由于 DStream 是由 RDD 组成的，从前面的章节中学到的技术和知识仍然适用于流式计算与实时应用。下一章我们来讲解如何运用 Spark 进行机器学习。

第 11 章基于 MLlib 的机器学习

MLlib 是 Spark 中提供机器学习函数的库。它是专为在集群上并行运行的情况而设计的。MLlib 中包含许多机器学习算法，可以在 Spark 支持的所有编程语言中使用。本章会展示如何在你的程序中调用 MLlib，并且给出一些常用的使用技巧。

机器学习本身是一个很大的话题，足够写很多本书，1 所以本章不会介绍机器学习本身。不过，如果你对机器学习很熟悉的话，你可以从本章学到如何在 Spark 中使用机器学习；即使你是机器学习的初学者，你也能够把本章的内容与其他关于机器学习的介绍性文字联系起来。如果你是有机器学习背景的数据科学家或者是与机器学习专家共事的工程师，并且正在尝试使用 Spark，那么你就应该读读本章内容。

> 1如 O'Reilly 出版的 *Machine Learning with R* 和 *Machine Learning for Hackers* 等。

11.1　概述

MLlib 的设计理念非常简单：把数据以 RDD 的形式表示，然后在分布式数据集上调用各种算法。MLlib 引入了一些数据类型（比如点和向量），不过归根结底，MLlib 就是 RDD 上一系列可供调用的函数的集合。比如，如果要用 MLlib 来完成文本分类的任务（例如识别垃圾邮件），你只需要按如下步骤操作。

(1) 首先用字符串 RDD 来表示你的消息。

(2) 运行 MLlib 中的一个特征提取（feature extraction）算法来把文本数据转换为数值特征（适合机器学习算法处理）；该操作会返回一个向量 RDD。

(3) 对向量 RDD 调用分类算法（比如逻辑回归）；这步会返回一个模型对象，可以使用该对象对新的数据点进行分类。

(4) 使用 MLlib 的评估函数在测试数据集上评估模型。

需要注意的是，MLlib 中只包含能够在集群上运行良好的并行算法，这一点很重要。有些经典的机器学习算法没有包含在其中，就是因为

它们不能并行执行。相反地，一些较新的研究得出的算法因为适用于集群，也被包含在 MLlib 中，例如分布式随机森林算法（distributed random forests）、K-means|| 聚类、交替最小二乘算法（alternating least squares）等。这样的选择使得 MLlib 中的每一个算法都适用于大规模数据集。如果你要在许多小规模数据集上训练各机器学习模型，最好还是在各节点上使用单节点的机器学习算法库（例如 Weka，http://www.cs.waikato.ac.nz/ml/weka/，或 SciKit-Learn，http://scikit-learn.org/stable/）实现，比如可以用 Spark 的 `map()` 操作在各节点上并行使用。类似地，我们在机器学习流水线中也常常用同一算法的不同参数对小规模数据集分别训练，来选出最好的一组参数。在 Spark 中，你可以通过把参数列表传给 `parallelize()` 来在不同的节点上分别运行不同的参数，而在每个节点上则使用单节点的机器学习库来实现。只有当你需要在一个大规模分布式数据集上训练模型时，MLlib 的优势才能突显出来。

最后，在 Spark 1.0 和 Spark 1.1 中，MLlib 的接口相对来说比较底层，提供给你的是不同任务应调用的函数，而不是高层的机器学习流水线所需要的工作流程（例如把输入数据分为训练数据和测试数据，或者尝试参数的多种组合）。在 Spark 1.2 中，MLlib 引入了流水线 API（在本书成书之际还是试验性功能 2），可以用来构建这样的流水线。这套 API 和类似 SciKit-Learn 这样的高层库很像，可以让人很容易地写出完整的、可自动调节的机器学习流水线。本章还是以低级 API 为主，我们会在本章最后简要介绍一下这套 API。

> 2在 1.4.0 中已成为正式接口。——译者注

11.2　系统要求

MLlib 需要你的机器预装一些线性代数的库。首先，你需要为操作系统安装 `gfortran` 运行库。如果 MLlib 警告找不到 `gfortran` 库的话，可以按 MLlib 网站（http://spark.apache.org/docs/latest/mllib-guide.html）上说明的步骤处理。其次，如果你要在 Python 中使用 MLlib，你需要安装 NumPy（http://www.numpy.org/）。如果你的 Python 没有安装 NumPy（即你无法使用 `import numpy`），最简单的办法就是使用 Linux 的包管理工具来安装 `python-numpy` 包或 `numpy` 包，或者使用第三方定制的 Python 版本，比如 Anaconda（http://continuum.io/downloads）。

MLlib 支持的算法也在随着时间推移不断发展。本章讨论的是 Spark

1.2 中 MLlib 支持的算法，有一些可能在早期的 Spark 版本中不存在。

11.3　机器学习基础

在开始讲 MLlib 中的函数之前，先来简单回顾一下机器学习的相关概念。

机器学习算法尝试根据训练数据（training data）使得表示算法行为的数学目标最大化，并以此来进行预测或作出决定。机器学习问题分为几种，包括分类、回归、聚类，每种都有不一样的目标。拿分类（classification）作为一个简单的例子：分类是基于已经被标记的其他数据点（比如一些已经分别被标记为垃圾邮件或非垃圾邮件的邮件）作为例子来识别一个数据点属于几个类别中的哪一种（比如判断一封邮件是不是垃圾邮件）。

所有的学习算法都需要定义每个数据点的特征（feature）集，也就是传给学习函数的值。举个例子，对于一封邮件来说，一些特征可能包括其来源服务器、提到 free 这个单词的次数、字体颜色等。在很多情况下，正确地定义特征才是机器学习中最有挑战性的部分。例如，在产品推荐的任务中，仅仅加上一个额外的特征（例如我们意识到推荐给用户的书籍可能也取决于用户看过的电影），就有可能极大地改进结果。

大多数算法都只是专为数值特征（具体来说，就是一个代表各个特征值的数字向量）定义的，因此提取特征并转化为特征向量是机器学习过程中很重要的一步。例如，在文本分类中（比如垃圾邮件和非垃圾邮件的例子），有好几个提取文本特征的方法，比如对各个单词出现的频率进行计数。

当数据已经成为特征向量的形式后，大多数机器学习算法都会根据这些向量优化一个定义好的数学函数。例如，某个分类算法可能会在特征向量的空间中定义出一个平面，使得这个平面能“最好”地分隔垃圾邮件和非垃圾邮件。这里需要为“最好”给出定义（比如大多数数据点都被这个平面正确分类）。算法会在运行结束时返回一个代表学习决定的模型（比如这个选中的平面），而这个模型就可以用来对新的点进行预测（例如根据新邮件的特征向量在平面的哪一边来决定它是不是垃圾邮件）。图 11-1 展示了一个机器学习流水线的示例。

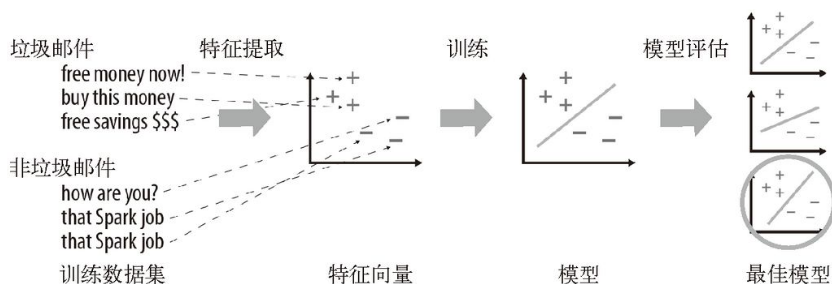


图 **11-1**：机器学习流水线中的典型步骤

最后，大多数机器学习算法都有多个会影响结果的参数，所以现实中的机器学习流水线会训练出多个不同版本的模型，然后分别对其进行评估（evaluate）。要这么做的话，通常需要把输入数据分为“训练集”和“测试集”，并且只使用前者进行训练，这样就可以用后者来检验模型是否过度拟合（overfit）了训练数据。MLlib 提供了几个算法来进行模型评估。

示例：垃圾邮件分类

举一个用来快速了解 MLlib 的例子，我们会展示一个构建垃圾邮件分类器的简单程序（见例 11-1 至例 11-3）。这个程序使用了 MLlib 中的两个函数：`HashingTF` 与 `LogisticRegressionWithSGD`，前者从文本数据构建词频（term frequency）特征向量，后者使用随机梯度下降法（Stochastic Gradient Descent，简称 SGD）实现逻辑回归。假设我们从两个文件 spam.txt 与 normal.txt 开始，两个文件分别包含垃圾邮件和非垃圾邮件的例子，每行一个。接下来我们就根据词频把每个文件中的文本转化为特征向量，然后训练出一个可以把两类消息分开的逻辑回归模型。代码和数据文件可以在本书的 Git 仓库中找到。

例 **11-1**：Python 版垃圾邮件分类器

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.feature import HashingTF
from pyspark.mllib.classification import LogisticRegressionWithSGD

spam = sc.textFile("spam.txt")
normal = sc.textFile("normal.txt")

# 创建一个HashingTF实例来把邮件文本映射为包含10000个特征的向量
tf = HashingTF(numFeatures = 10000)
# 各邮件都被切分为单词，每个单词被映射为一个特征
```

```
spamFeatures = spam.map(lambda email: tf.transform(email.split(" ")))
normalFeatures = normal.map(lambda email: tf.transform(email.split(" ")))

# 创建LabeledPoint数据集分别存放阳性（垃圾邮件）和阴性（正常邮件）的例子
positiveExamples = spamFeatures.map(lambda features: LabeledPoint(1, featu
negativeExamples = normalFeatures.map(lambda features: LabeledPoint(0, fea
trainingData = positiveExamples.union(negativeExamples)
trainingData.cache() # 因为逻辑回归是迭代算法，所以缓存训练数据RDD

# 使用SGD算法运行逻辑回归
model = LogisticRegressionWithSGD.train(trainingData)

# 以阳性（垃圾邮件）和阴性（正常邮件）的例子分别进行测试。首先使用
# 一样的HashingTF特征来得到特征向量，然后对该向量应用得到的模型
posTest = tf.transform("O M G GET cheap stuff by sending money to ...".spl
negTest = tf.transform("Hi Dad, I started studying Spark the other ...".sp
print "Prediction for positive test example: %g" % model.predict(posTest)
print "Prediction for negative test example: %g" % model.predict(negTest)
```

例 11-2：Scala 版垃圾邮件分类器

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.feature.HashingTF
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD

val spam = sc.textFile("spam.txt")
val normal = sc.textFile("normal.txt")

// 创建一个HashingTF实例来把邮件文本映射为包含10000个特征的向量
val tf = new HashingTF(numFeatures = 10000)
// 各邮件都被切分为单词，每个单词被映射为一个特征
val spamFeatures = spam.map(email => tf.transform(email.split(" ")))
val normalFeatures = normal.map(email => tf.transform(email.split(" ")))

// 创建LabeledPoint数据集分别存放阳性（垃圾邮件）和阴性（正常邮件）的例子
val positiveExamples = spamFeatures.map(features => LabeledPoint(1, featur
val negativeExamples = normalFeatures.map(features => LabeledPoint(0, feat
val trainingData = positiveExamples.union(negativeExamples)
trainingData.cache() // 因为逻辑回归是迭代算法，所以缓存训练数据RDD

// 使用SGD算法运行逻辑回归
val model = new LogisticRegressionWithSGD().run(trainingData)

// 以阳性（垃圾邮件）和阴性（正常邮件）的例子分别进行测试
val posTest = tf.transform(
  "O M G GET cheap stuff by sending money to ...".split(" "))
val negTest = tf.transform(
  "Hi Dad, I started studying Spark the other ...".split(" "))
println("Prediction for positive test example: " + model.predict(posTest))
println("Prediction for negative test example: " + model.predict(negTest))
```

例 11-3：Java 版垃圾邮件分类器

```
import org.apache.spark.mllib.classification.LogisticRegressionModel;
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD;
import org.apache.spark.mllib.feature.HashingTF;
import org.apache.spark.mllib.linalg.Vector;
import org.apache.spark.mllib.regression.LabeledPoint;

JavaRDD<String> spam = sc.textFile("spam.txt");
JavaRDD<String> normal = sc.textFile("normal.txt");

// 创建一个HashingTF实例来把邮件文本映射为包含10000个特征的向量
final HashingTF tf = new HashingTF(10000);

// 创建LabeledPoint数据集分别存放阳性（垃圾邮件）和阴性（正常邮件）的例子
JavaRDD<LabeledPoint> posExamples = spam.map(new Function<String, LabeledP
  public LabeledPoint call(String email) {
    return new LabeledPoint(1, tf.transform(Arrays.asList(email.split(" ")
  }
});
JavaRDD<LabeledPoint> negExamples = normal.map(new Function<String, Labele
  public LabeledPoint call(String email) {
    return new LabeledPoint(0, tf.transform(Arrays.asList(email.split(" ")
  }
});
JavaRDD<LabeledPoint> trainData = positiveExamples.union(negativeExamples)
trainData.cache(); // 因为逻辑回归是迭代算法，所以缓存训练数据RDD

// 使用SGD算法运行逻辑回归
LogisticRegressionModel model = new LogisticRegressionWithSGD().run(trainD

// 以阳性（垃圾邮件）和阴性（正常邮件）的例子分别进行测试
Vector posTest = tf.transform(
  Arrays.asList("O M G GET cheap stuff by sending money to ...".split(" ")
Vector negTest = tf.transform(
  Arrays.asList("Hi Dad, I started studying Spark the other ...".split(" "
System.out.println("Prediction for positive example: " + model.predict(pos
System.out.println("Prediction for negative example: " + model.predict(neg
```

如你所见，这段代码在各种语言中都很相似。代码直接操作 RDD——在本例中，所操作的是字符串 RDD（即原始文本）和 `LabeledPoints`（MLlib 中用来存放特征向量和对应标签的数据类型）RDD。

11.4　数据类型

MLlib 包含一些特有的数据类型，它们位于

org.apache.spark.mllib 包（Java/Scala）或 pyspark.mllib（Python）内。主要的几个如下所列。

- **Vector**

  一个数学向量。MLlib 既支持稠密向量也支持稀疏向量，前者表示向量的每一位都存储下来，后者则只存储非零位以节约空间。后面会简单讨论不同种类的向量。向量可以通过 mllib.linalg.Vectors 类创建出来。

- **LabeledPoint**

  在诸如分类和回归这样的监督式学习（supervised learning）算法中，LabeledPoint 用来表示带标签的数据点。它包含一个特征向量与一个标签（由一个浮点数表示），位置在 mllib.regression 包中。

- **Rating**

  用户对一个产品的评分，在 mllib.recommendation 包中，用于产品推荐。

- 各种**Model**类

  每个 Model 都是训练算法的结果，一般有一个 predict() 方法可以用来对新的数据点或数据点组成的 RDD 应用该模型进行预测。

大多数算法直接操作由 Vector、LabeledPoint 或 Rating 对象组成的 RDD。你可以用任意方式创建出这些对象，不过一般来说你需要通过对外部数据进行转化操作来构建出 RDD——例如，通过读取一个文本文件或者运行一条 Spark SQL 命令。接下来，使用 map() 将你的数据对象转为 MLlib 的数据类型。

操作向量

作为 MLlib 中最常用的数据类型，Vector 类有一些需要注意的地方。

第一，向量有两种：稠密向量与稀疏向量。稠密向量把所有维度的值存放在一个浮点数数组中。例如，一个 100 维的向量会存储 100 个双精度浮点数。相比之下，稀疏向量只把各维度中的非零值存储下

来。当最多只有 10% 的元素为非零元素时，我们通常更倾向于使用稀疏向量（不仅是出于对内存使用的考虑，也是出于对速度的考虑）。许多特征提取技术都会生成非常稀疏的向量，所以这种方式常常是一种很关键的优化手段。

第二，创建向量的方式在各种语言中有一些细微的差别。在 Python 中，你在 MLlib 中任意地方传递的 NumPy 数组都表示一个稠密向量，你也可以使用 `mllib.linalg.Vectors` 类创建其他类型的向量（如例 11-4 所示）。**3** 而在 Java 和 Scala 中，都需要使用 `mllib.linalg.Vectors` 类（如例 11-5 和例 11-6 所示）。

> 3如果你使用的是 SciPy，Spark 也能够将大小为 $N\times 1$ 的 `scipy.sparse` 矩阵识别为长度为 N 的向量。

例 11-4：用 Python 创建向量

```
from numpy import array
from pyspark.mllib.linalg import Vectors

# 创建稠密向量<1.0, 2.0, 3.0>
denseVec1 = array([1.0, 2.0, 3.0]) # NumPy数组可以直接传给MLlib
denseVec2 = Vectors.dense([1.0, 2.0, 3.0]) # 或者使用Vectors类来创建

# 创建稀疏向量<1.0, 0.0, 2.0, 0.0>；该方法只接收
# 向量的维度（4）以及非零位的位置和对应的值
# 这些数据可以用一个dictionary来传递，或使用两个分别代表位置和值的list
sparseVec1 = Vectors.sparse(4, {0: 1.0, 2: 2.0})
sparseVec2 = Vectors.sparse(4, [0, 2], [1.0, 2.0])
```

例 11-5：用 Scala 创建向量

```
import org.apache.spark.mllib.linalg.Vectors

// 创建稠密向量<1.0, 2.0, 3.0>；Vectors.dense接收一串值或一个数组
val denseVec1 = Vectors.dense(1.0, 2.0, 3.0)
val denseVec2 = Vectors.dense(Array(1.0, 2.0, 3.0))

// 创建稀疏向量<1.0, 0.0, 2.0, 0.0>；该方法只接收
// 向量的维度（这里是4）以及非零位的位置和对应的值
val sparseVec1 = Vectors.sparse(4, Array(0, 2), Array(1.0, 2.0))
```

例 11-6：用 Java 创建向量

```
import org.apache.spark.mllib.linalg.Vector;
```

```
import org.apache.spark.mllib.linalg.Vectors;

// 创建稠密向量<1.0, 2.0, 3.0>；Vectors.dense接收一串值或一个数组
Vector denseVec1 = Vectors.dense(1.0, 2.0, 3.0);
Vector denseVec2 = Vectors.dense(new double[] {1.0, 2.0, 3.0});

// 创建稀疏向量<1.0, 0.0, 2.0, 0.0>；该方法只接收
// 向量的维度（这里是4）以及非零位的位置和对应的值
Vector sparseVec1 = Vectors.sparse(4, new int[] {0, 2}, new double[]{1.0,
```

最后，在 Java 和 Scala 中，MLlib 的 `Vector` 类只是用来为数据表示服务的，而没有在用户 API 中提供加法和减法这样的向量的算术操作。（在 Python 中，你可以对稠密向量使用 NumPy 来进行这些数学操作，也可以把这些操作传给 MLlib。）这主要是为了让 MLlib 保持在较小规模内，因为实现一套完整的线性代数库超出了本工程的范围。如果你想在你的程序中进行向量的算术操作，可以使用一些第三方的库，比如 Scala 中的 Breeze（https://github.com/scalanlp/breeze）或者 Java 中的 MTJ（https://github.com/fommil/matrix-toolkits-java），然后再把数据转为 MLlib 向量。

11.5　算法

本节会介绍 MLlib 中主要的算法，以及它们的输入和输出类型。我们没有足够的篇幅来解释各个算法的数学原理，因此只能专注于如何调用并配置这些算法。

11.5.1　特征提取

`mllib.feature` 包中包含一些用来进行常见特征转化的类。这些类中有从文本（或其他表示）创建特征向量的算法，也有对特征向量进行正规化和伸缩变换的方法。

TF-IDF

词频—逆文档频率（简称 TF-IDF）是一种用来从文本文档（例如网页）中生成特征向量的简单方法。它为文档中的每个词计算两个统计值：一个是词频（TF），也就是每个词在文档中出现的次数，另一个是逆文档频率（IDF），用来衡量一个词在整个文档语料库中出现的（逆）频繁程度。这些值的积，也就是 TF × IDF，展示了一个词与特定文档的相关程度（比如这个词在某文档中很常见，但在整个语料库中却很少见）。

MLlib 有两个算法可以用来计算 TF-IDF：`HashingTF` 和 `IDF`，都在 `mllib.feature` 包内。`HashingTF` 从一个文档中计算出给定大小的词频向量。为了将词与向量顺序对应起来，它使用了哈希法（hasing trick）。在类似英语这样的语言中，有几十万个单词，因此将每个单词映射到向量中的一个独立的维度上需要付出很大代价。而 `HashingTF` 使用每个单词对所需向量的长度 *S* 取模得出的哈希值，把所有单词映射到一个 0 到 *S*-1 之间的数字上。由此我们可以保证生成一个 *S* 维的向量。在实践中，即使有多个单词被映射到同一个哈希值上，算法依然适用。MLlib 开发者推荐将 *S* 设置在 218 到 220 之间。

`HashingTF` 可以一次只运行于一个文档中，也可以运行于整个 RDD 中。它要求每个“文档”都使用对象的可迭代序列来表示——例如 Python 中的 `list` 或 Java 中的 `Collection`。

例 11-7 在 Python 中使用了 `HashingTF`。

例 11-7：在 Python 中使用 `HashingTF`

```
>>> from pyspark.mllib.feature import HashingTF

>>> sentence = "hello hello world"
>>> words = sentence.split() # 将句子切分为一串单词
>>> tf = HashingTF(10000) # 创建一个向量，其尺寸S = 10,000
>>> tf.transform(words)
SparseVector(10000, {3065: 1.0, 6861: 2.0})

>>> rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())
>>> tfVectors = tf.transform(rdd) # 对整个RDD进行转化操作
```

 在真实流水线中，你可能需要在把文档传给 TF 之前，对文档进行预处理并提炼单词。例如，你可能需要把所有的单词转为小写、去除标点、去除 ing 这样的后缀。为了得到最佳结果，你可以在 `map()` 中调用一些类似 NLTK（http://www.nltk.org/）这样的单节点自然语言处理库。

当你构建好词频向量之后，你就可以使用 `IDF` 来计算逆文档频率，然后将它们与词频相乘来计算 TF-IDF。你首先要对 `IDF` 对象调用 `fit()` 方法来获取一个 `IDFModel`，它代表语料库中的逆文档频率。接下来，对模型调用 `transform()` 来把 TF 向量转为 IDF 向量。例 11-8 展示了如何从例 11-7 中计算 IDF。

例 **11-8**：在 Python 中使用 TF-IDF

```
from pyspark.mllib.feature import HashingTF, IDF

# 将若干文本文件读取为TF向量
rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())
tf = HashingTF()
tfVectors = tf.transform(rdd).cache()

# 计算IDF，然后计算TF-IDF向量
idf = IDF()
idfModel = idf.fit(tfVectors)
tfIdfVectors = idfModel.transform(tfVectors)
```

注意，我们对 RDD`tfVectors` 调用了 `cache()` 方法，因为它被使用了两次（一次是训练 IDF 模型时，一次是用 IDF 乘以 TF 向量时）。

1. 缩放

大多数机器学习算法都要考虑特征向量中各元素的幅值，并且在特征缩放调整为平等对待时表现得最好（例如所有的特征平均值为 0，标准差为 1）。当构建好特征向量之后，你可以使用 MLlib 中的 StandardScaler 类来进行这样的缩放，同时控制均值和标准差。你需要创建一个 `StandardScaler`，对数据集调用 `fit()` 函数来获取一个 `StandardScalerModel`（也就是为每一列计算平均值和标准差），然后使用这个模型对象的 `transform()` 方法来缩放一个数据集，如例 11-9 所示。

例 **11-9**：在 Python 中缩放向量

```
from pyspark.mllib.feature import StandardScaler

vectors = [Vectors.dense([-2.0, 5.0, 1.0]), Vectors.dense([2.0, 0.0, 1.0])]
dataset = sc.parallelize(vectors)
scaler = StandardScaler(withMean=True, withStd=True)
model = scaler.fit(dataset)
result = model.transform(dataset)

# 结果：{[-0.7071, 0.7071, 0.0], [0.7071, -0.7071, 0.0]}
```

2. 正规化

在一些情况下，在准备输入数据时，把向量正规化为长度 1 也是有用

的。使用 `Normalizer` 类可以实现，只要使用 `Normalizer.transform(rdd)` 就可以了。默认情况下，`Normalizer` 使用 L^2 范式（也就是欧几里得距离），不过你可以给 `Normalizer` 传递一个参数 p 来使用 L^p 范式。

3. Word2Vec

Word2Vec（https://code.google.com/p/word2vec/）4是一个基于神经网络的文本特征化算法，可以用来将数据传给许多下游算法。Spark 在 `mllib.feature.Word2Vec` 类中引入了该算法的一个实现。

> 4引用 Mikolov 等人所写文章“Efficient Estimation of Word Representations in Vector Space,”2013。

要训练 Word2Vec，你需要传给它一个用 `String` 类（每个单词用一个）的 `Iterable` 表示的语料库。和前面的“TF-IDF”一节所讲的很像，Word2Vec 也推荐对单词进行正规化处理（例如全部转为小写、去除标点和数字）。当你训练好模型（通过 `Word2Vec.fit(rdd)`）之后，你会得到一个 `Word2VecModel`，塔可以用来将每个单词通过 `transform()` 转为一个向量。注意，Word2Vec 算法中模型的大小等于你的词库中的单词数乘以向量的大小（向量大小默认为 100）。你可能希望筛选掉不在标准字典中的单词来控制模型大小。一般来说，比较合适的词库大小约为 100 000 个词。

11.5.2　统计

不论是在即时的探索中，还是在机器学习的数据理解中，基本的统计都是数据分析的重要部分。MLlib 通过 `mllib.stat.Statistics` 类中的方法提供了几种广泛使用的统计函数，这些函数可以直接在 RDD 上使用。一些常用的函数如下所列。

- `Statistics.colStats(`*`rdd`*`)`

  计算由向量组成的 RDD 的统计性综述，保存着向量集合中每列的最小值、最大值、平均值和方差。这可以用来在一次执行中获取丰富的统计信息。

- `Statistics.corr(`*`rdd, method`*`)`

计算由向量组成的 RDD 中的列间的相关矩阵，使用皮尔森相关（Pearson correlation）或斯皮尔曼相关（Spearman correlation）中的一种（*method* 必须是 `pearson` 或 `spearman` 中的一个）。

- `Statistics.corr(`*`rdd1, rdd2, method`*`)`

  计算两个由浮点值组成的 RDD 的相关矩阵，使用皮尔森相关或斯皮尔曼相关中的一种（*method* 必须是 `pearson` 或 `spearman` 中的一个）。

- `Statistics.chiSqTest(`*`rdd`*`)`

  计算由 `LabeledPoint` 对象组成的RDD中每个特征与标签的皮尔森独立性测试（Pearson's independence test）结果。返回一个 `ChiSqTestResult` 对象，其中有 p 值（p-value）、测试统计及每个特征的自由度。标签和特征值必须是分类的（即离散值）。

除了这些算法以外，数值 RDD 还提供几个基本的统计函数，例如 `mean()`、`stdev()` 以及 `sum()`，我们在 6.6 节中有所提及。除此以外，RDD 还支持 `sample()` 和 `sampleByKey()`，使用它们可以构建出简单而分层的数据样本。

11.5.3　分类与回归

分类与回归是监督式学习的两种主要形式。监督式学习指算法尝试使用有标签的训练数据（也就是已知结果的数据点）根据对象的特征预测结果。分类和回归的区别在于预测的变量的类型：在分类中，预测出的变量是离散的（也就是一个在有限集中的值，叫作类别）；比如，分类可能是将邮件分为垃圾邮件和非垃圾邮件，也有可能是文本所使用的语言。在回归中，预测出的变量是连续的（例如根据年龄和体重预测一个人的身高）。

分类和回归都会使用 MLlib 中的 `LabeledPoint` 类。11.4 节中提过，这个类在 `mllib.regression` 包中。一个 `LabeledPoint` 其实就是由一个 `label`（`label` 总是一个 `Double` 值，不过可以为分类算法设为离散整数）和一个 `features` 向量组成。



对于二元分类，MLlib 预期的标签为 0 或 1。在某些情况下可能会使用 -1 和 1，但这会导致错误的

结果。对于多元分类，MLlib 预期标签范围是从 0 到 C-1，其中 C 表示类别数量。

MLlib 包含多种分类与回归的算法，其中包括简单的线性算法以及决策树和森林算法。

1. 线性回归

线性回归是回归中最常用的方法之一，是指用特征的线性组合来预测输出值。MLlib 也支 持 L^1 和 L^2 的正则的回归，通常称为 Lasso 和 ridge 回归。

线性回归算法可以使用的类包括 `mllib.regression.LinearRegressionWithSGD`、`LassoWithSGD` 以及 `RidgeRegressionWithSGD`。这遵循了 MLlib 中常用的命名模式，即对于涉及多个算法的问题，在类名中使用“With”来表明所使用的算法。这里，SGD 代表的是随机梯度下降法。

这些类都有几个可以用来对算法进行调优的参数。

- `numIterations`

  要运行的迭代次数（默认值：`100`）。

- `stepSize`

  梯度下降的步长（默认值：`1.0`）。

- `intercept`

  是否给数据加上一个干扰特征或者偏差特征——也就是一个值始终为 1 的特征（默认值：false）。

- `regParam`

  Lasso 和 ridge 的正规化参数（默认值：1.0）。

调用算法的方式在不同语言中略有不同。在 Java 和 Scala 中，你需要创建一个 `LinearRegressionWithSGD` 对象，调用它的 setter 方法来设置参数，然后调用 `run()` 来训练模型。在 Python 中，你需要使用类的方法 `LinearRegressionWithSGD.train()`，并对

其传递键值对参数。在这两种情况中，你都需要传递一个由 `LabeledPoint` 组成的 RDD，如例 11-10 至例 11-12 所示。

例 **11-10**：Python 中的线性回归

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.regression import LinearRegressionWithSGD

points = # (创建LabeledPoint组成的RDD)
model = LinearRegressionWithSGD.train(points, iterations=200, intercept=Tr
print "weights: %s, intercept: %s" % (model.weights, model.intercept)
```

例 **11-11**：Scala 中的线性回归

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.regression.LinearRegressionWithSGD

val points: RDD[LabeledPoint] = // ...
val lr = new LinearRegressionWithSGD().setNumIterations(200).setIntercept
val model = lr.run(points)
println("weights: %s, intercept: %s".format(model.weights, model.intercept
```

例 **11-12**：Java 中的线性回归

```
import org.apache.spark.mllib.regression.LabeledPoint;
import org.apache.spark.mllib.regression.LinearRegressionWithSGD;
import org.apache.spark.mllib.regression.LinearRegressionModel;

JavaRDD<LabeledPoint> points = // ...
LinearRegressionWithSGD lr =
  new LinearRegressionWithSGD().setNumIterations(200).setIntercept(true);
LinearRegressionModel model = lr.run(points.rdd());
System.out.printf("weights: %s, intercept: %s\n",
  model.weights(), model.intercept());
```

注意，在 Java 中，需要通过调用 `.rdd()` 方法把 JavaRDD 转为 Scala 中的 `RDD` 类。这种模式在 MLlib 中随处可见，因为 MLlib 方法被设计为既可以用 Java 调用，也可以用 Scala 调用。

一旦训练完成，所有语言中返回的 `LinearRegressionModel` 都会包含一个 `predict()` 函数，可以用来对单个特征向量预测一个值。`RidgeRegressionWithSGD` 和 `LassoWithSGD` 的行为类似，并且也会返回一个类似的模型类。事实上，这种通过 setter 方法

调节算法参数，然后返回一个带有 `predict()` 方法的 `Model` 对象的模式，在 MLlib 中很常见。

2. 逻辑回归

逻辑回归是一种二元分类方法，用来寻找一个分隔阴性和阳性示例的线性分割平面。在 MLlib 中，它 接收一组标签为 0 或 1 的 `LabeledPoint`，返回可以预测新点的分类的 `LogisticRegressionModel` 对象。

逻辑回归算法的 API 和前小一节中讲的线性回归十分相似。两者的区别之一在于，有两 种可以用来解决逻辑回归问题的算法：SGD 和 LBFGS5。LBFGS 一般是最好的选择，但是 在早期的 MLlib 版本（早于 Spark 1.2）中不可用。这些算法通过 `mllib.classification.LogisticRegressionWithLBFGS` 和 `WithSGD` 类提供给用户，接口和 `LinearRegressionWithSGD` 相似。它们接收的参数和线性回归完全一样（详见前一小节）。

> 5LBFGS 是牛顿法的近似，它可以在比随机梯度下降更少的迭代次数内收敛。详见 http://en.wikipedia.org/wiki/Limited-memory_BFGS。

这两个算法中得出的 `LogisticRegressionModel` 可以为每个点求出一个在 0 到 1 之间的得分，之后会基于一个阈值返回 `0` 或 `1`：默认情况下，对于 0.5，它会返回 `1`。你可以通过 `setThreshold()` 改变阈值，也可以通过 `clearThreshold()` 去除阈值设置，这样的话 `predict()` 就会返回原始得分。对于阴性阳性示例各半的均衡数据集，我们推荐保留 0.5 作为阈值。对于不平衡的数据集，你可以通过提升阈值来减少假阳性数据的数量（也就是提高精确率，但是也降低了召回率），也可以通过降低阈值来减少假阴性数据的数量。


在使用逻辑回归时，将特征提前缩放到相同范围内通常比较重要。你可以使用 MLlib 的 StandardScaler 来实现特征缩放，如 11.5.1 节中的“缩放”小节所述。

3. 支持向量机

支持向量机（简称 SVM）算法是另一种使用线性分割平面的二元分类算法，同样只预期 0 或者 1 的标签。通过 `SVMWithSGD` 类，我们可以访问这种算法，它的参数与线性回归和逻辑回归的参数差不多。返

回的 `SVMModel` 与 `LogisticRegressionModel` 一样使用阈值的方式进行预测。

4. 朴素贝叶斯

朴素贝叶斯（Naive Bayes）算法是一种多元分类算法，它使用基于特征的线性函数计算将一个点分到各类中的得分。这种算法通常用于使用 TF-IDF 特征的文本分类，以及其他一些应用。MLlib 实现了多项朴素贝叶斯算法，需要非负的频次（比如词频）作为输入特征。

在 MLlib 中，你可以通过 `mllib.classification.NaiveBayes` 类来使用朴素贝叶斯算法。它支持一个参数 `lambda`（Python 中是 `lambda_`），用来进行平滑化。你可以对一个由 `LabeledPoint` 组成的 RDD 调用朴素贝叶斯算法，对于 C 个分类，标签值范围在 0 至 C-1 之间。

返回的 `NaiveBayesModel` 让我们可以使用 `predict()` 预测对某点最合适的分类，也可以访问训练好的模型的两个参数：各特征与各分类的可能性矩阵 `theta`（对于 C 个分类和 D 个特征的情况，矩阵大小为 $C \times D$），以及表示先验概率的 C 维向量 `pi`。

5. 决策树与随机森林

决策树是一个灵活的模型，可以用来进行分类，也可以用来进行回归。决策树以节点树的形式表示，每个节点基于数据的特征作出一个二元决定（比如，这个人的年龄是否大于 20 ？），而树的每个叶节点则包含一种预测结果（例如，这个人是不是会买一个产品？）。决策树的吸引力在于模型本身容易检查，而且决策树既支持分类的特征，也支持连续的特征。图 11-2 展示了一个决策树的示例。

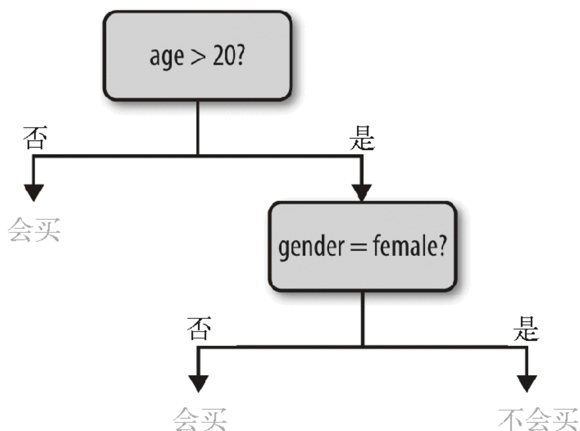


图 **11-2**：一个预测用户是否会购买一件产品的决策树示例

在 MLlib 中，你可以使用 `mllib.tree.DecisionTree` 类中的静态方法 `trainClassifier()` 和 `trainRegressor()` 来训练决策树。和其他有些算法不同的是，Java 和 Scala 的 API 也使用静态方法，而不使用 setter 方法定制的 `DecisionTree` 对象。该训练方法接收如下所列参数。

- `data`

  由 `LabeledPoint` 组成的 RDD。

- `numClasses`（仅用于分类时）

  要使用的类别数量。

- `impurity`

  节点的不纯净度测量；对于分类可以为 `gini` 或 `entropy`，对于回归则必须为 `variance`。

- `maxDepth`

  树的最大深度（默认值：`5`）。

- `maxBins`

  在构建各节点时将数据分到多少个箱子中（推荐值：`32`）。

- `categoricalFeaturesInfo`

  一个映射表，用来指定哪些特征是分类的，以及它们各有多少个分类。例如，如果特征 1 是一个标签为 0 或 1 的二元特征，特征 2 是一个标签为 0、1 或 2 的三元特征，你就应该传递 `{1: 2, 2: 3}`。如果没有特征是分类的，就传递一个空的映射表。

MLlib 的在线文档（http://spark.apache.org/docs/latest/mllib-decision-tree.html）中包含了对此处所使用算法的详细解释。算法的开销会随训练样本数目、特征数量以及 `maxBins` 参数值进行线性增长。对于大规模数据集，你可能需要使用较低的 `maxBins` 值来更快地训练模型，尽管这也会降低算法的质量。

`train()` 方法会返回一个 `DecisionTreeModel` 对象。你可以使用这个对象的 `predict()` 方法来对一个新的特征向量预测对应的值，或者预测一个向量 RDD。你也可以使用 `toDebugString()` 来输出这棵树。这个对象是可序列化的，所以你可以用 Java 序列化将它保存，然后在另一个程序中读取出来。

最后，在 Spark 1.2 中，MLlib 在 Java 和 Scala 中添加了试验性的 `RandomForest` 类，可以用来构建一组树的组合，也被称为随机森林。它可以通过 `RandomForest.trainClassifier` 和 `trainRegressor` 使用。除了刚才列出的每棵树对应的参数外，`RandomForest` 还接收如下参数。

- `numTrees`

  要构建的树的数量。提高 `numTrees` 可以降低对训练数据过度拟合的可能性。

- `featureSubsetStrategy`

  在每个节点上作决定时需要考虑的特征数量。可以是 `auto`（让库来自动选择）、`all`、`sqrt`、`log2` 以及 `onethird`；越大的值所花费的代价越大。

- `seed`

  所使用的随机数种子。

随机森林算法返回一个 `WeightedEnsembleModel` 对象，其中包

含几个决策树（在 weakHypotheses 字段中，权重由 weakHypothesisWeights 决定），可以对 RDD 或 vector 调用 predict()。它还有一个 toDebugString 方法，可以打印出其中所有的树。

11.5.4 聚类

聚类算法是一种无监督学习任务，用于将对象分到具有高度相似性的聚类中。前面提到的监督式任务中的数据都是带标签的，而聚类可以用于无标签的数据。该算法主要用于数据探索（查看一个新数据集是什么样子）以及异常检测（识别与任意聚类都相距较远的点）。

KMeans

MLlib 包含聚类中流行的 K-means 算法，以及一个叫作 K-means|| 的变种，可以为并行环境提供更好的初始化策略。**6**K-means|| 的初始化过程与 K-means++ 在配置单节点时所进行的初始化过程非常相似。

> **6**最早介绍 K-means|| 的文章是 Bahmani 等人所著的“Scalable K-Means++,”VLDB 2008。

K-means 中最重要的参数是生成的聚类中心的目标数量 K。事实上，你几乎不可能提前知道聚类的“真实”数量，所以最佳实践是尝试几个不同的 K 值，直到聚类内部平均距离不再显著下降为止。然而，算法一次只能接收一个 K 值。除了 K 以外，MLlib 中的 K-means 还接收以下几个参数。

- `initializationMode`

  用来初始化聚类中心的方法，可以是“k-means||”或者“random”；k-means||（默认值）一般会带来更好的结果，但是开销也会略高一些。

- `maxIterations`

  运行的最大迭代次数（默认值：100）。

- `runs`

  算法并发运行的数目。MLlib 的 K-means 算法支持从多个起点并发执行，然后选择最佳结果，这也是获取较好的整体模型的

一种不错的方法（K-means 的运行可以停止在本地最小值上）。

和其他算法一样，当你要调用 K-means 算法时，你需要创建 `mllib.clustering.KMeans` 对象（在 Java/Scala 中）或者调用 `KMeans.train`（在 Python 中）。它接收一个 `Vector` 组成的 RDD 作为参数。K-means 返回一个 `KMeansModel` 对象，该对象允许你访问其 `clusterCenters` 属性（聚类中心，是一个向量的数组）或者调用 `predict()` 来对一个新的向量返回它所属的聚类。注意，`predict()` 总是返回和该点距离最近的聚类中心，即使这个点跟所有的聚类都相距很远。

11.5.5　协同过滤与推荐

协同过滤是一种根据用户对各种产品的交互与评分来推荐新产品的推荐系统技术。协同过滤吸引人的地方就在于它只需要输入一系列用户 / 产品的交互记录：无论是“显式”的交互（例如在购物网站上进行评分）还是“隐式”的（例如用户访问了一个产品的页面但是没有对产品评分）交互皆可。仅仅根据这些交互，协同过滤算法就能够知道哪些产品之间比较相似（因为相同的用户与它们发生了交互）以及哪些用户之间比较相似，然后就可以作出新的推荐。

尽管 MLlib 的 API 使用了“用户”和“产品”的概念，但你也可以将协同过滤用于其他应用场景中，比如在社交网络中推荐用户，为文章推荐要添加的标签，为电台推荐歌曲等。

交替最小二乘

MLlib 中包含交替最小二乘（简称 ALS）的一个实现，这是一个协同过滤的常用算法，可以很好地扩展到集群上。7 它位于 `mllib.recommendation.ALS` 类中。

> 7两篇对网络规模数据的 ALS 算法的研究论文分别是 Zhou 等人撰写的“Large-Scale Parallel Collaborative Filtering for the Netflix Prize”和 Hu 等人撰写的“Collaborative Filtering for Implicit Feedback Datasets,”都发表于 2008 年。

ALS 会为每个用户和产品都设一个特征向量，这样用户向量与产品向量的点积就接近于他们的得分。它接收下面所列这些参数。

- `rank`

使用的特征向量的大小；更大的特征向量会产生更好的模型，但是也需要花费更大的计算代价（默认值：10）。

- iterations

  要执行的迭代次数（默认值：10）。

- lambda

  正则化参数（默认值：0.01）。

- alpha

  用来在隐式 ALS 中计算置信度的常量（默认值：1.0）。

- numUserBlocks，numProductBlocks

  切分用户和产品数据的块的数目，用来控制并行度；你可以传递 -1 来让 MLlib 自动决定（默认行为）。

要使用 ALS 算法，你需要有一个由 mllib.recommendation.Rating 对象组成的 RDD，其中每个包含一个用户 ID、一个产品 ID 和一个评分（要么是显式的评分，要么是隐式反馈；参见接下来的讨论）。实现过程中的一个挑战是每个 ID 都需要是一个 32 位的整型值。如果你的 ID 是字符串或者更大的数字，我们推荐你直接在 ALS 中使用 ID 的哈希值；即使有两个用户或者两个产品映射到同一个 ID 上，总体结果依然会不错。还有一种办法是 broadcast() 一张从产品 ID 到整型值的表，来赋给每个产品独特的 ID。

ALS 返回一个 MatrixFactorizationModel 对象来表示结果，可以调用 predict() 来对一个由 (userID, productID) 对组成的 RDD 进行预测评分。8 你也可以使用 model.recommendProducts (userId, numProducts) 来为一个给定用户找到最值得推荐的前 numProduct 个产品。注意，和 **MLlib** 中的其他模型不同，**MatrixFactorizationModel** 对象很大，为每个用户和产品都存储了一个向量。这样我们就不能把它存到磁盘上，然后在另一个程序中读取回来。不过，你可以把模型中生成的特征向量 RDD，也就是 model.userFeatures 和 model.productFeatures 保存到分布式文件系统上。

8在 Java 中，从一个由 Tuple2 组成的 JavaRDD 开始，需要先对它调用 .rdd() 方

法。

最后，ALS 有两个变种：显式评分（默认情况）和隐式反馈（通过调用 `ALS.trainImplicit()` 而非 `ALS.train()` 来打开）。用于显式评分时，每个用户对于一个产品的评分需要是一个得分（例如 1 到 5 星），而预测出来的评分也是得分。而用于隐式反馈时，每个评分代表的是用户会和给定产品发生交互的置信度（比如随着用户访问一个网页次数的增加，评分也会提高），预测出来的也是置信度。关于对隐式反馈使用 ALS 算法，Hu 等人所撰写的“Collaborative Filtering for Implicit Feedback Datasets,”ICDM 2008 中有更为详细的介绍。

11.5.6　降维

1. 主成分分析

给定一个高维空间中的点的数据集，我们经常需要减少点的维度，来使用更简单的工具对其进行分析。例如，我们可能想要在二维平面上画出这些点，或者只是想减少特征的数量使得模型训练更加高效。

机器学习社区中使用的主要的降维技术是主成分分析（简称 PCA，https://en.wikipedia.org/wiki/Principal_component_analysis）。在这种技术中，我们会把特征映射到低维空间，让数据在低维空间表示的方差最大化，从而忽略一些无用的维度。要计算出这种映射，我们要构建出正规化的相关矩阵，并使用这个矩阵的奇异向量和奇异值。与最大的一部分奇异值相对应的奇异向量可以用来重建原始数据的主要成分。

PCA 目前只在 Java 和 Scala（MLlib 1.2）中可用。要调用 PCA，你首先要使用 `mllib.linalg.distributed.RowMatrix` 类来表示你的矩阵，然后存储一个由 `Vector` 组成的 RDD，每行一个。**9** 之后，你就可以如例 11-13 所示那样调用 PCA 算法。

9在 Java 中，从一个由 `Vector` 组成的 JavaRDD 开始，我们需要对它调用 `.rdd()` 方法把它转为一个 Scala 的 RDD。

例 **11-13**：Scala 中的 PCA

```
import org.apache.spark.mllib.linalg.Matrix
import org.apache.spark.mllib.linalg.distributed.RowMatrix
```

```
val points: RDD[Vector] = // ...
val mat: RowMatrix = new RowMatrix(points)
val pc: Matrix = mat.computePrincipalComponents(2)

// 将点投影到低维空间中
val projected = mat.multiply(pc).rows

// 在投影出的二维数据上训练k-means模型
val model = KMeans.train(projected, 10)
```

在这个例子中，投影出的 RDD 包含原始 RDD 中的 `points` 的二维版本，可以用来作图或进行其他 MLlib 算法，比如使用 K-means 进行聚类。

注意，`computePrincipalComponents()` 返回的是 `mllib.linalg.Matrix` 对象，是一个和 `Vector` 相似的表示稀疏矩阵的工具类。你可以调用 `toArray` 方法获取底层的数据。

2. 奇异值分解

MLlib 也提供了低层的奇异值分解（简称 SVD）原语。SVD 会把一个 $m \times n$ 的矩阵 A 分解成三个矩阵 $A \approx U\Sigma V^{\mathrm{T}}$，其中：

- U 是一个正交矩阵，它的列被称为左奇异向量；
- Σ 是一个对角线上的值均为非负数并降序排列的对角矩阵，它的对角线上的值被称为奇异值；
- V 是一个正交矩阵，它的列被称为右奇异向量。

对于大型矩阵，通常不需要进行完全分解，只需要分解出靠前的奇异值和与之对应的奇异向量即可。这样可以节省存储空间、降噪，并有利于恢复低秩矩阵。如果保留前 k 个奇异值，那么结果矩阵就会是 $U : m \times k$，$\Sigma : k \times k$ 以及 $V : n \times k$。

要进行分解，应调用 `RowMatrix` 类的 `computeSVD` 方法，如例 11-14 所示。

例 **11-14**：Scala 中的 SVD

```
// 计算RowMatrix矩阵的前20个奇异值及其对应的奇异向量
val svd: SingularValueDecomposition[RowMatrix, Matrix] =
  mat.computeSVD(20, computeU=true)
val U: RowMatrix = svd.U // U是一个分布式RowMatrix
```

```
val s: Vector = svd.s     // 奇异值用一个局部稠密向量表示
val V: Matrix = svd.V     // V是一个局部稠密矩阵
```

11.5.7　模型评估

无论机器学习任务使用的是何种算法，模型评估都是端到端机器学习流水线的一个重要环节。许多机器学习任务可以使用不同的模型来应对，而且即使使用的是同一个算法，参数设置也可以带来不同的结果。不仅如此，我们还要考虑模型对训练数据过度拟合的风险，因此你最好通过在另一个数据集上测试模型来对模型进行评估，而不是使用训练数据集。

在本书写作时（对应于 Spark 1.2），MLlib 包含一组试验性模型评估函数，不过只能在 Java 和 Scala 中使用。这些函数在 `mllib.evaluation` 包中，根据问题的不同，它们在 `BinaryClassificationMetrics` 和 `MulticlassMetrics` 等这些不同的类中。使用这些类，你可以从由（预测，事实）对组成的 RDD 上创建出一个 `Metrics` 对象，然后计算诸如精确率、召回率、接受者操作特性（ROC）曲线下的面积等指标。这些方法应该运行在一个非训练集的测试集上（比如在训练前先划出 20% 的数据）。你可以使用 `map()` 函数将你的模型应用到测试数据集上，生成由（预测，事实）对组成的 RDD。

在将来的 Spark 版本中，本章末尾所介绍的流水线 API 有望对所有语言引入评估函数。通过流水线 API，你可以定义一个机器学习算法流水线以及一个评估标准，然后系统就可以使用交叉验证自动搜索参数并选择出最好的模型。

11.6　一些提示与性能考量

11.6.1　准备特征

尽管机器学习演讲中经常着重强调所使用的算法，但切记在实践中，每个算法的好坏只取决于你所使用的特征！许多从事大规模数据机器学习的人员都认为特征准备是大规模机器学习中最重要的一步。添加信息更丰富的特征（例如与其他数据集连接以引入更多信息）与将现有特征转为合适的向量表示（例如缩放向量）都能极大地帮助改进结果。

本书将不会对特征准备作全面讨论，但是我们鼓励你参考其他机器学习书籍以了解更多信息。不过，有以下这些通用的提示值得注意，尤其是对 MLlib 来说。

- 缩放输入特征。像 11.5.1 节的“伸缩”小节中讲解的那样，使用 `StandardScaler` 处理特征来平等对待特征。
- 正确提取文本特征。使用诸如 NLTK（http://www.nltk.org）这样的外部库来提词，在使用 TF-IDF 提取特征时，在有代表性的语料库上使用 IDF。
- 为分类标上正确的标签。MLlib 要求分类的标签是 0 到 $C-1$ 之间，其中 C 表示分类的总数。

11.6.2 配置算法

在正规化选项可用时，MLlib 中的大多数算法都会在正则化打开时表现得更好（在预测准确度方面）。此外，大多数基于 SGD 的算法需要大约 100 轮迭代来获得较好的结果。MLlib 尝试提供合适的默认值，但是你应该尝试增加迭代次数，来看看是否能够提高精确度。例如，使用 ALS 算法时，`rank` 的默认值 10 相对较低，所以你应该尝试提高这个值。确保在评估这些参数变化时将测试数据排除在训练集之外。

11.6.3 缓存RDD以重复使用

MLlib 中的大多数算法都是迭代的，对数据进行反复操作。因此，在把输入数据集传给 MLlib 前使用 `cache()` 将它缓存起来是很重要的。即使数据在内存中放不下，你也应该尝试 `persist(StorageLevel.DISK_ONLY)`。

在 Python 中，MLlib 会把数据集在从 Python 端传到 Java 端时在 Java 端自动缓存，因此没有必要缓存你的 Python RDD，除非你在自己的程序中还要用到它。而在 Scala 和 Java 中，则需要由你来决定是否执行缓存操作。

11.6.4 识别稀疏程度

当你的特征向量包含很多零时，用稀疏格式存储这些向量会为大规模数据集节省巨大的时间和空间。在空间方面，当至多三分之二的位为非零值时，MLlib 的稀疏表示比它的稠密表示要小。在数据处理代价

方面，当至多 10% 的位为非零值时，稀疏向量所要花费的代价也会更小。（这是因为使用稀疏表示需要对向量中的每个元素执行的指令比使用稠密向量表示时要多。）但是如果使用稀疏表示能够让你缓存使用稠密表示时无法缓存的数据，即使数据本身比较稠密，你也应当选择稀疏表示。

11.6.5　并行度

对于大多数算法而言，你的输入 RDD 的分区数至少应该和集群的 CPU 核心数相当，这样才能达到完全的并行。回想一下，默认情况下 Spark 会为文件的每个“块”创建一个分区，而块一般为 64 MB。你可以通过向 `SparkContext.textFile()` 这样的函数传递分区数的最小值来改变默认行为——例如 `sc.textFile("data.txt", 10)`。另一种方法是对 RDD 调用 `repartition(numPartitions)` 来将 RDD 分区成 `numPartitions` 个分区。你始终可以通过 Spark 的网页用户界面看到每个 RDD 的分区数。同时，注意不要使用太多分区，因为这会增加通信开销。

11.7　流水线API

从 Spark 1.2 起，基于机器学习流水线的概念，MLlib 增加了一套新的高层机器学习 API。这套 API 和 SciKit-Learn（http://scikit-learn.org）中提供的流水线 API 比较相似。简单地说，流水线就是一系列转化数据集的算法（要么是特征转化，要么是模型拟合）。流水线的每个步骤都可能有参数（例如逻辑回归中的迭代次数）。流水线 API 通过使用所选的评估矩阵评估各个集合，使用网格搜索自动找到最佳的参数集。

流水线 API 使用我们第 9 章中介绍过的 Spark SQL 中的 SchemaRDD 作为统一的数据集表示形式。SchemaRDD 中有多个有名字的列，这样要引用数据的不同字段就会比较容易。流水线的各步骤可能会给 SchemaRDD 加上新的列（例如提取了特征的数据）。总体的理念也和 R 中的 DataFrame 有些类似。

为了让你感受一下这套 API，我们把本章之前用过的垃圾邮件分类的例子用这种接口重新实现了一遍。我们也展示了如何通过对 `HashingTF` 和 `LogisticRegression` 的参数使用几组不同值并进行网格搜索，来让这个示例程序更加清晰明了，如例 11-15 所示。

例 **11-15**：在 Scala 中使用流水线 API 实现垃圾邮件分类

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.ml.Pipeline
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.feature.{HashingTF, Tokenizer}
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator

// 用来表示文档的类，会被转入SchemaRDD中
case class LabeledDocument(id: Long, text: String, label: Double)
val documents = // （读取LabeledDocument的RDD）

val sqlContext = new SQLContext(sc)
import sqlContext._

// 配置该机器学习流水线中的三个步骤：分词、词频计数、逻辑回归；每个步骤
// 会输出SchemaRDD的一个列，并作为下一个步骤的输入列
val tokenizer = new Tokenizer() // 把各邮件切分为单词
  .setInputCol("text")
  .setOutputCol("words")
val tf = new HashingTF() // 将邮件中的单词映射为包含10000个特征的向量
  .setNumFeatures(10000)
  .setInputCol(tokenizer.getOutputCol)
  .setOutputCol("features")
val lr = new LogisticRegression() // 默认使用"features"作为输入列
val pipeline = new Pipeline().setStages(Array(tokenizer, tf, lr))

// 使用流水线对训练文档进行拟合
val model = pipeline.fit(documents)

// 或者，不使用上面的参数只对训练集进行一次拟合，也可以

// 通过交叉验证对一批参数进行网格搜索，来找到最佳的模型
val paramMaps = new ParamGridBuilder()
  .addGrid(tf.numFeatures, Array(10000, 20000))
  .addGrid(lr.maxIter, Array(100, 200))
  .build() // 构建参数的所有组合
val eval = new BinaryClassificationEvaluator()
val cv = new CrossValidator()
  .setEstimator(lr)
  .setEstimatorParamMaps(paramMaps)
  .setEvaluator(eval)
val bestModel = cv.fit(documents)
```

在写作本书时，流水线 API 还属于试验性接口，你可以从 MLlib 文档（http://spark.apache.org/docs/latest/mllib-guide.html）中找到关于流水线 API 的最新文档。

11.8　总结

本章概述了 Spark 的机器学习算法库。如你所见，MLlib 与 Spark 的其他 API 紧密联系。它可以让你操作 RDD，而得到的结果也可以在其他 Spark 函数中使用。MLlib 也是 Spark 开发最为活跃的组件之一，它还在不断发展中。我们建议查阅你所使用的 Spark 版本所对应的官方文档（http://spark.apache.org/documentation.html）以了解其中的最新功能。

作者简介

Holden Karau 是 Databricks 的软件开发工程师，开源工作积极参与者。她也是早前另一本 Spark 书的作者。在加入 Databricks 之前，她曾在 Google、Foursquare、Amazon 参与过搜 索和分类问题方面的工作。Holden 毕业于滑铁卢大学，获得计算机科学专业的数学学士学 位。除了软件外，她还喜爱玩火、焊接和呼啦圈。

Andy Konwinski 是 Databricks 的创始人之一。在此之前，他是加州大学伯克利分校 AMPLab 实验室的博士生，接着成为了博士后，研究方向是大规模分布式计算和集群调度。他共同 创建了 Apache Mesos 项目，并且是该项目的代码提交者之一。他还与 Google 的系统工程师 以及研究员一起致力于 Omega——Google 的下一代集群调度系统的设计。最近，他开展并 领导了 AMP Camp 大数据训练营以及 Spark 峰会，并为 Spark 项目作出了贡献。

Patrick Wendell 也是 Databricks 的联合创始人之一，同时他也是一位 Spark 的代码提交者及 PMC 成员。在 Spark 项目中，Patrick 是几个 Spark 发布版本的发行经理，其中包括 Spark 1.0。Patrick 也维护着 Spark 核心引擎的几个子系统。在帮助创办 Databricks 之前，Patrick 在加州大学伯克利分校获得了计算机科学的硕士学位，研究方向是大规模分析类工作负载 的低延迟调度。他还拥有普林斯顿大学的计算机学士学位。

Matei Zaharia 是 Apache Spark 的创造者，也是 Databricks 的 CTO。他拥有加州大学伯克利 分校的博士学位，并从那里以研究型项目的形式启动了 Spark。他现在也是 Apache 基金会 的一名副总裁。除了 Spark 以外，他也对集群计算领域的其他一些项目有所研究，并作出 了开源代码共献，其中包括 Apache Hadoop（他是代码提交者之一）和 Apache Mesos（也 是他在伯克利时参与启动的项目）。

封面介绍

本书封面上的动物是斑点猫鲨（Scyliorhinus canicula），是东北大西洋和地中海中最常见的 软骨鱼类之一。这是一种体型小而修长的鲨鱼，头部扁钝，眼睛细长，吻部短圆。背部表 面呈灰棕色，混杂着细小的或明或暗的斑点图案。皮肤质地粗糙，和砂纸的粗糙度相似。

这种小鲨鱼以海生无脊椎动物为食，它的食物包括软体动物、甲壳类、头足类，以及多毛 类蠕虫。它也会吃一些小的硬骨鱼，偶尔吃体型稍大的鱼。它是一个卵生物种，会把蛋产 在靠近海岸的浅水中，由带有长卷须的角质壳保护。

斑点猫鲨在渔场中具有一定的商业价值，但它更适合用来在公共水族馆中展示。尽管它的 商业价值已被发现，且大量个体被保留下来供人食用，但这一物种仍然经常被抛弃，而且 研究表明抛弃后的存活率较高。 O'Reilly 丛书封面上的许多动物都濒临灭绝，而它们对这个世界来说都很重要。要了解更 多你力所能及的事，请访问 animals.oreilly.com。

封面图片来自 Wood 所著 *Animate Creation*。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_interview，讲述码农精彩人生
- 微信 图灵教育：turingbooks

091507240605ToBeReplacedWithUserId

版权信息

书名：数据科学入门

作者：[美] Joel Grus

译者：高蓉　韩波

ISBN：978-7-115-41741-1

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

091507240605ToBeReplacedWithUserId

版权声明
O'Reilly Media, Inc. 介绍
业界评论
前言
数据科学
从零开始
本书排版约定
示例代码的使用
Safari® Books Online
联系我们
致谢
第 1 章　导论
1.1　数据的威力
1.2　什么是数据科学
1.3　激励假设：DataSciencester
1.3.1　寻找关键联系人
1.3.2　你可能知道的数据科学家
1.3.3　工资与工作年限
1.3.4　付费账户
1.3.5　兴趣主题
1.4　展望
第 2 章　Python 速成
2.1　基础内容
2.1.1　Python获取
2.1.2　Python之禅
2.1.3　空白形式
2.1.4　模块
2.1.5　算法
2.1.6　函数
2.1.7　字符串
2.1.8　异常
2.1.9　列表

2.1.10 元组
2.1.11 字典
2.1.12 集合
2.1.13 控制流
2.1.14 真和假
2.2 进阶内容
2.2.1 排序
2.2.2 列表解析
2.2.3 生成器和迭代器
2.2.4 随机性
2.2.5 正则表达式
2.2.6 面向对象的编程
2.2.7 函数式工具
2.2.8 枚举
2.2.9 压缩和参数拆分
2.2.10 args和kwargs
2.2.11 欢迎来到DataSciencester
2.3 延伸学习
第 3 章 可视化数据
3.1 matplotlib
3.2 条形图
3.3 线图
3.4 散点图
3.5 延伸学习
第 4 章 线性代数
4.1 向量
4.2 矩阵
4.3 延伸学习
第 5 章 统计学
5.1 描述单个数据集
5.1.1 中心倾向
5.1.2 离散度

5.2　相关
5.3　辛普森悖论
5.4　相关系数其他注意事项
5.5　相关和因果
5.6　延伸学习
第 6 章　概率
6.1　不独立和独立
6.2　条件概率
6.3　贝叶斯定理
6.4　随机变量
6.5　连续分布
6.6　正态分布
6.7　中心极限定理
6.8　延伸学习
第 7 章　假设与推断
7.1　统计假设检验
7.2　案例：掷硬币
7.3　置信区间
7.4　P-hacking
7.5　案例：运行A/B测试
7.6　贝叶斯推断
7.7　延伸学习
第 8 章　梯度下降
8.1　梯度下降的思想
8.2　估算梯度
8.3　使用梯度
8.4　选择正确步长
8.5　综合
8.6　随机梯度下降法
8.7　延伸学习
第 9 章　获取数据
9.1　stdin和stdout

9.2　读取文件
9.2.1　文本文件基础
9.2.2　限制的文件
9.3　网络抓取
9.3.1　HTML和解析方法
9.3.2　案例：关于数据的O'Reilly图书
9.4　使用API
9.4.1　JSON（和XML）
9.4.2　使用无验证的API
9.4.3　寻找API
9.5　案例：使用Twitter API
获取证明文件
9.6　延伸学习
第 10 章　数据工作
10.1　探索你的数据
10.1.1　探索一维数据
10.1.2　二维数据
10.1.3　多维数据
10.2　清理与修改
10.3　数据处理
10.4　数据调整
10.5　降维
10.6　延伸学习
第 11 章　机器学习
11.1　建模
11.2　什么是机器学习
11.3　过拟合和欠拟合
11.4　正确性
11.5　偏倚-方差权衡
11.6　特征提取和选择
11.7　延伸学习
第 12 章　k 近邻法

12.1　模型
12.2　案例：最喜欢的编程语言
12.3　维数灾难
12.4　延伸学习
第 13 章　朴素贝叶斯算法
13.1　一个简易的垃圾邮件过滤器
13.2　一个复杂的垃圾邮件过滤器
13.3　算法的实现
13.4　测试模型
13.5　延伸学习
第 14 章　简单线性回归
14.1　模型
14.2　利用梯度下降法
14.3　最大似然估计
14.4　延伸学习
第 15 章　多重回归分析
15.1　模型
15.2　最小二乘模型的进一步假设
15.3　拟合模型
15.4　解释模型
15.5　拟合优度
15.6　题外话：Bootstrap
15.7　回归系数的标准误差
15.8　正则化
15.9　延伸学习
第 16 章　逻辑回归
16.1　问题
16.2　Logistic函数
16.3　应用模型
16.4　拟合优度
16.5　支持向量机
16.6　延伸学习

第 17 章　决策树
17.1　什么是决策树
17.2　熵
17.3　分割之熵
17.4　创建决策树
17.5　综合运用
17.6　随机森林
17.7　延伸学习
第 18 章　神经网络
18.1　感知器
18.2　前馈神经网络
18.3　反向传播
18.4　实例：战胜CAPTCHA
18.5　延伸学习
第 19 章　聚类分析
19.1　原理
19.2　模型
19.3　示例：聚会
19.4　选择聚类数目k
19.5　示例：对色彩进行聚类
19.6　自下而上的分层聚类
19.7　延伸学习
第 20 章　自然语言处理
20.1　词云
20.2　n-grams模型
20.3　语法
20.4　题外话：吉布斯采样
20.5　主题建模
20.6　延伸学习
第 21 章　网络分析
21.1　中介中心度
21.2　特征向量中心度

21.2.1　矩阵乘法
21.2.2　中心度
21.3　有向图与PageRank
21.4　延伸学习
第 22 章　推荐系统
22.1　手工甄筛
22.2　推荐流行事物
22.3　基于用户的协同过滤方法
22.4　基于物品的协同过滤算法
22.5　延伸学习
第 23 章　数据库与 SQL
23.1　CREATE TABLE与INSERT
23.2　UPDATE
23.3　DELETE
23.4　SELECT
23.5　GROUP BY
23.6　ORDER BY
237　JOIN
23.8　子查询
23.9　索引
23.10　查询优化
23.11　NoSQL
23.12　延伸学习
第 24 章　MapReduce
24.1　案例：单词计数
24.2　为什么是MapReduce
24.3　更加一般化的MapReduce
24.4　案例：分析状态更新
24.5　案例：矩阵计算
24.6　题外话：组合器
24.7　延伸学习
第 25 章　数据科学前瞻

25.1　IPython
25.2　数学
25.3　不从零开始
25.3.1　NumPy
25.3.2　pandas
25.3.3　scikit-learn
25.3.4　可视化
25.3.5　R
25.4　寻找数据
25.5　从事数据科学
25.5.1　Hacker News
25.5.2　消防车
25.5.3　T恤
25.5.4　你呢？
作者简介
关于封面

版权声明

© 2015 by O'Reilly Media, Inc.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2016. Authorized translation of the English edition, 2015 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2015。

简体中文版由人民邮电出版社出版，2016。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 **O'Reilly** 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，

并且切实地按照 **Yogi Berra** 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，**Tim** 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

前言

数据科学

有人称数据科学家为“21世纪头号性感职业”（https://hbr.org/2012/10/data-scientist-the-sexiest-job-of-the-21st-century/）。虽说如此称呼有些夸张，但这个名称对数据科学的推崇却一点也没错，这是一个蓬勃发展、前途无限的行业。很多分析师都预言，未来十年会需要比现在多得多的数据科学工作者。

那么，什么是数据科学？唯有正确理解数据科学，才能培养出数据科学家。根据广受业界赞誉的文氏图（http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram），数据科学是以下几个方面的交叉：

- 黑客技能
- 数学和统计学知识
- 专业技能

我原本很想写一本能涵盖以上三个方面的书，但很快意识到仅关于专业技能的撰写就会耗费上万页笔墨，于是及时放弃转而专注于前两个方面。我的目标有两个：一是帮助读者掌握从事数据科学工作所必需的黑客技能；二是帮助读者熟悉数学和统计学，这是数据科学的核心。

对一本书来说，这两个愿望有点大了。学习黑客技能的最好方法就是钻研技术。通过阅读本书，你可以理解我钻研技术的方式，但相同的方式对你未必最适合；你可以理解我使用的一些工具，但相同的工具对你来说未必最顺手；你可以理解我如何解决数据问题，但相同的方式对你来说未必最有效。举例的目的和希望是启发你以自己的方式和方法完成工作。本书涵盖的所有代码和数据都可以从 GitHub 上下载。

同样，学习数学的最好方式就是研习数学。当然本书并不是一部数学著作，我们在本书中大半也不会“研习数学”，我想强调的是数学知识对从事数据科学工作至关重要。不理解概率、统计、线性代数，就无

法真正开始数据科学工作。在需要的地方，书中会引入数学方程式、数学直觉、数学公理，以及借以阐释大数学思想的卡通漫画。有我在，别怕！

总之，数据科学相当有趣（尤其和税务筹划或者煤矿开采等其他工作相比）。

从零开始

很多很多的数据科学库、框架、模块、工具箱可以有效地实现数据科学大部分常见的（和不常见的）算法与技术。如果你是一位数据科学家，就会非常熟悉 NumPy、scikit-learn、pandas 以及其他库。这些库对数据科学工作至关重要。如果还没有真正理解数据科学，运用这些库也是开始数据科学工作的好方式。

在本书中，我们从零开始着手数据科学工作。这意味着为了获得更好的理解，我们需要自己亲手构建工具和实现算法。我花费了很多心思选择注释良好、简洁易读的实现范例。在大部分情形下，所建立的工具意义清晰但实用性有限，它们对规模较小的示例数据集运转良好，但对“网络级别”的数据集就束手无策了。

在全书中，我会向读者指出相应的库，用以将相应技术运用于大规模数据集，但本书中我们不会使用它们。

对学习数据科学，一直有这样一种积极的争论，即什么样的语言环境最好？许多人认为是统计语言 R。（我们说，他们错了。）还有一些人认为是 Java 或者 Scala。而我认为，Python 才是最佳选择！

对于学习和从事数据科学工作，Python 具有几大优势：

- 免费；
- 编程相对简单（尤其是也易于理解）；
- 具有很多数据科学相关的库。

我不敢说 Python 是我最爱的编程语言，因为的确存在其他一些更舒适、设计更棒、编程更有乐趣的语言。但是，每当着手一个新的数据科学项目时，我最终使用的是 Python；每当需要快速构建某个有效程序的原型时，我使用的是 Python；每当需要用简洁易懂的方式表达数据科学概念时，我使用的还是 Python。于是，本书也采用

Python。

但是，教授 Python 不是本书的目的（尽管通过学习本书你会学到一些 Python 知识）。本书会用一章快速介绍 Python 的重要特征，这些特征与本书目的紧密相关。倘若读者没有 Python 基础（或编程基础），那需要再补充阅读一些关于 Python 的入门指导。

本书数据科学导论的其余部分采取了类似的书写方式，在必要或需要阐明时才深入细节，否则省略细节留给读者自己去挖掘（或者在维基百科上查阅）。

过去我曾培训过许多数据科学家。不是每个人都会努力变成改变世界的明星级数据忍者，但所有人都通过培训成为了更棒的数据科学家。我越来越相信，任何拥有一定数学基础和编程技术的人，只要再匹配一些基本材料就可以从事数据科学工作。必需品是好奇心、勤奋工作的态度，还有本书。没错，就是本书！

本书排版约定

本书使用了下列排版约定。

- 楷体

  表示新术语。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 等宽粗体（**`constant width bold`**）

  表示应该由用户输入的命令或其他文本。

- 等宽斜体（*`constant width italic`*）

  表示应该由用户输入的值或根据上下文确定的值替换的文本。



该图标表示提示或建议。



该图标表示一般注释。



该图标表示警告或警示。

示例代码的使用

本书的补充材料（示例代码、练习等）都可以从 GitHub 下载：https://github.com/joelgrus/data-science-from-scratch。

本书提供代码的目的是帮你快速完成工作。一般情况下，你可以在你的程序或文档中使用本书中的代码，而不必取得我们的许可，除非你想复制书中很大一部分代码。例如，你在编写程序时，用到了本书中的几个代码片段，这不必取得我们的许可。但若将 O'Reilly 图书中的代码制成光盘并进行出售或传播，则需获得我们的许可。引用示例代码或书中内容来解答问题无需许可。将书中很大一部分的示例代码用于你个人的产品文档，这需要我们的许可。

如果你引用了本书的内容并标明版权归属声明，我们对此表示感谢，但这不是强制的。版权归属声明通常包括：标题、作者、出版社和 ISBN，例如：“*Data Science from Scratch* by Joel Grus (O'Reilly). Copyright 2015 Joel Grus, 978-1-4919-0142-7”。

如果你认为你对示例代码的使用已经超出上述范围，或者你对是否需要获得示例代码的授权还不清楚，请随时联系我们：permissions@oreilly.com 。

Safari® Books Online



Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体（https://www.safaribooksonline.com/enterprise/）、政府机构（https://www.safaribooksonline.com/government/）、教育机构（https://www.safaribooksonline.com/academic-public-library/）和个人，Safari Books Online 提供各种产品组合和灵活的定价策略（https://www.safaribooksonline.com/pricing/）。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社（https://www.safaribooksonline.com/our-library/）的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920033400.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

首先，我深深地感谢 Mike Loukides 接受我写作本书的提议（并对本书的篇幅提出了合理的建议）。其实他可以选择更轻松的方式，例如可以说：“那个总发样章给我的是什么人？我该怎样拒绝他？”但他没有，我感激不尽！同样，我很感谢我的编辑 Marie Beaugureau，她在整个出版过程中给予我指导，并最终使本书呈现出比我独立完成更棒的状态。

如果我从未学习数据科学，怎么能写出这样一本书？如果没有 Dave Hsu、Igor Tatarinov、John Rauser 和 Farecast 群组其他人的影响，我不大可能学习数据科学（当时甚至还没有数据科学这个名称）。还要感谢免费大型公开课程项目 Coursera 的优秀教师们。

同样，我很感谢本书的试读者和评论者。Jay Fundling 找出了许多错误，并指出很多含混的表达，因而本书得到了很大的改善，非常感谢。Debashis Ghosh 全面检查了本书的统计学部分。本书原本表达了肯定 Python 否定 R 的看法，Andrew Musselman 建议淡化这种立场，我后来体会到这是金玉良言。我也非常感谢 Trey Causey、Ryan Matthew Balfanz、Loris Mularoni、Núria Pujol、Rob Jefferson、Mary Pat Campbell、Zach Geary 和 Wendy Grus 给我的宝贵反馈。当然，本书其余的问题，责任在我。

我非常感谢 Twitter 上的数据科学社区，它让我接触到很多新奇的概念，让我认识了很多大牛。我深感自己的欠缺，需要写一本书来弥补。（再次）特别感谢 Trey Causey，他（并非刻意地）提醒我在书中加一章线性代数的内容。同样感谢 Sean J. Taylor，他（并非刻意

地）指出了数据处理一章中的若干重大缺漏。

最后，向 Ganga 和 Madeline 致以我无限的感恩。比写一本书更痛苦的事，莫过于和写这本书的人朝夕相对。本书得以完成，全赖家人的支持与鼓励。

第 1 章　导论

“数据！数据！！数据！！！”他不耐烦地咆哮着，“我不能做无米之炊！”

——阿瑟 · 柯南 · 道尔

1.1　数据的威力

生活中，数据无处不在。用户的每次点击，网站都会记录下来。你每时每刻的位置和速度，智能手机也会记录下来。“量化自我”生活方式的倡导者使用智能计步器记录心率、行动习惯、饮食习惯、睡眠方式。智能汽车记录驾驶习惯，智能家居设施记录生活习惯，智能购物设备记录购物习惯，等等。互联网是一个广袤的知识谱系，包括有无数交叉引用的百科全书，电影、音乐、赛讯、弹球机、模因、鸡尾酒等各种专业数据库，以及许多政府发布的多得让人理不清头绪的统计数据（某些还是比较真实的）。

在这些数据之中隐藏着无数问题的答案，这些问题从没有人提出过。让我们在这本书中一起学习如何找出这些问题。

1.2　什么是数据科学

有这样一个形容数据科学家的小笑话。数据科学家有什么特点呢？他们是计算机科学家中的统计专家，是统计专家中的计算机科学家（抱歉，笑话有点冷）。事实上，有些数据科学家的确是统计专家，而有些数据科学家则堪比软件工程师。有的数据科学家是机器学习专家，而也有一些数据科学家仅仅是这方面的菜鸟。有的数据科学家拥有博士学位，出版过出色的学术作品，而有些数据科学家从不阅读论文（我都有点不好意思了）。总之，不管你如何定义数据科学家，总会找到某些反例来否定这样的定义。

然而，这并不会阻止我们尝试为数据科学家下定义。我们认为，数据科学家是能够从混乱数据中剥离出洞见的人。今天，世界各地有无数人着力于此。

举个例子，一个名叫 OkCupid 的约会网站为了给会员找到合适的对

象，要求他们回答上千个问题。OkCupid 通过分析这些答案来确定你可以问哪些无伤大雅的问题，以此来猜测你和某个心仪之人初次约会时会有多大可能直接奔向三垒（http://blog.okcupid.com/index.php/the-best-questions-for-first-dates/）。

再举个例子，你在 Facebook 上需要填写家乡和居住地的信息。表面上看，网站是在帮助你的朋友更容易地找到你，联系你。但实际上，除此以外，网站还通过分析地理信息来研究全球移民模式（https://www.facebook.com/notes/facebook-data-science/coordinated-migration/10151930946453859），或者研究不同球队的粉丝分布情况（https://www.facebook.com/notes/facebook-data-science/nfi-fans-on-facebook/10151298370823859）。

另一个例子，一个名叫 Target 的大型零售商，会追踪消费者在线上线下的购买与互动数据（http://www.nytimes.com/2012/02/19/magazine/shopping-habits.html），再从中分析，预测哪些客户怀孕了，以便向其推销合适的母婴商品。

最后，举一个奥巴马 2012 年竞选的例子。他的竞选团队雇用了很多数据科学家，专家们搜集选民的相关数据，通过数据挖掘识别不同的选民。他们通过实验的方法确定哪些选民需要更多的关注，并选择最有鼓舞性的拉票活动，把重心放在最能吸引选票的活动上。最后，奥巴马胜出，成功地第二次出任总统。人们普遍认为数据科学家功不可没。同时，这意味着数据分析在未来竞选中会扮演越来越重要的角色，一场数据竞争的硝烟弥漫开来，好戏刚刚开始。

现在，如果谈论竞争令你感到疲惫，那我们转而谈谈公益——数据科学家偶尔也会使用数据来帮助政府提高工作效率（http://www.marketplace.org/topics/tech/beyond-ad-clicks-using-big-data-social-good）、帮助流浪者（http://dssg.io/2014/08/20/paths-homelessness.html）、改善公共健康（https://plus.google.com/communities/109572103057302114737）等。当然，如果你能想出好方法来提高广告点击率，这对你的职业发展也不会有什么坏处。

1.3　激励假设：DataSciencester

恭喜！你被聘请来领导 DataSciencester 的数据科学工作。DataSciencester 是数据科学家们的社交网络。

虽然号称为数据科学家服务，DataSciencester 却从未践行数据科学任

务（同样也从未构建自己的产品）。当然，这是你的工作。在本书中，我们通过解决在工作中碰到的一个个问题，来学习数据科学的思想。我们有时会直接研究用户提供的数据，有时会研究用户与网站互动生成的数据，有时研究从我们自己设计的实验中获得的数据。

DataSciencester 拥有一种特别强烈的原创精神——“非我莫属”（Not-Invented-Here，NIH），即工作中使用到的工具必须自己亲手创建。这样完成工作之后，你会全面深入地理解数据科学。将来，你更可以将自己所学运用于更有前途的公司，或去解决任何有趣的问题。

欢迎加入，祝你好运！（周五可以穿牛仔裤上班，洗手间位于大厅右侧。）

1.3.1　寻找关键联系人

现在，你在 DataSciencester 开始第一天的工作。网络部的副总一直有一些关于客户的问题没有解决，以前找不到人帮忙，现在你来了，他特别高兴。

第一，他需要你识别出数据科学家中的“关键联系人”。因而他转给你 DataSciencester 所有用户的网络关系数据。（实际工作中，你需要的数据不会轻而易举地拿过来。我们在第 9 章专门讨论获取数据的方法。）

从整体上看，数据是一个包含所有用户的列表。列表的每个元素是一个字典（即一个 `dict`）。字典中包含了用户的 ID（即 `id`）和账号名（即 `name`。在这批数据中，`name` 与 `id` 的数字为谐音）：

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
]
```

同时，他也给了你用户的“友邻关系”数据列表。这个列表的元素是成对的 `id`：

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (5,
```

比如说，元组 (0, 1) 表示 id 为 0 的数据科学家 Hero 和 id 为 1 的数据科学家 Dunn 是朋友。这种网络关系可以用图 1-1 来描述。

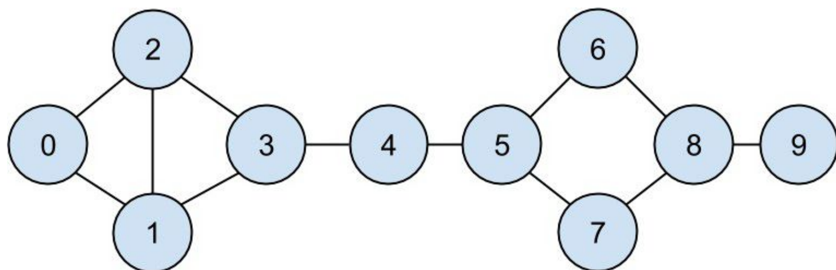


图 **1-1**：**DataSciencester** 网络

我们使用字典结构 dict 表示用户，因而可以方便地添加更多数据。



目前不要纠结代码细节，我们会在第 2 章指导你快速学习 Python。现在你只需要大致了解代码的功能。

比如，如果我们希望对每个用户增加一个朋友列表，首先需要对每个用户创建一个空列表：

```
for user in users:
    user["friends"] = []
```

再用 friendships 数据填充：

```
for i, j  in friendships:
    # 这能起作用是因为users[i]是id为i的用户
    users[i]["friends"].append(users[j]) # 把i加为j的朋友
    users[j]["friends"].append(users[i]) # 把j加为i的朋友
```

完成之后，每个用户的 dict 都会包含一个朋友列表，进而可以深入地研究朋友网络关系，比如提问“平均的联系数是多少”（即每位用户平均拥有几位朋友）。

首先计算出全部的联系数，这需要对所有用户的 friends 列表的长

度求和：

```
def number_of_friends(user):
    """how many friends does _user_have?"""
    return len(user["friends"])                    # 列表friend_ids的长度

total_connections = sum(number_of_friends(user)
                        for user in users)         # 24
```

然后，将它除以用户个数：

```
from __future__ import division                  # 整数除法需要导入
num_users = len(users)                           # 列表users的长度
avg_connections = total_connections / num_users  # 2.4
```

这样就很容易看出，拥有最多联系的人就是拥有最多朋友数目的人。

因为用户不多，所以能很方便地按照朋友数的多少排序：

```
# 创建一个列表(user_id, number_of_friends)
num_friends_by_id = [(user["id"], number_of_friends(user))
                     for user in users]

sorted(num_friends_by_id,                                # 把它按照
       key=lambda (user_id, num_friends): num_friends,   # num_friends
       reverse=True)                                     # 从最大到最小排序

# 每一对都是(user_id, num_friends)
# [(1, 3), (2, 3), (3, 3), (5, 3), (8, 3),
#  (0, 2), (4, 2), (6, 2), (7, 2), (9, 1)]
```

可将以上行为视为一种识别谁处在人际网络中心的方法。事实上，以上计算的是度中心性，是一种网络度量，如图 1-2 所示。

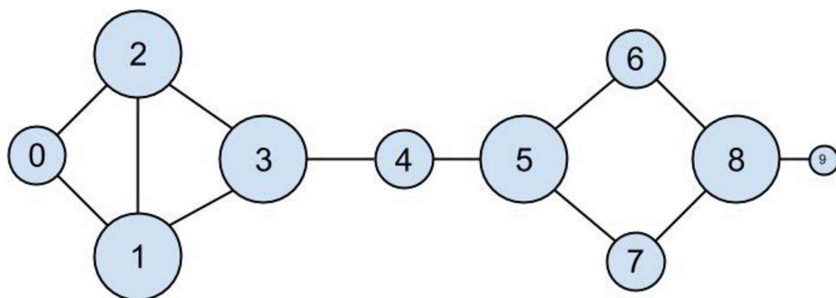


图 **1-2**：利用度计算 **DataSciencester** 的网络大小

度中心性简单易算，但不能总如你所愿。比如，在 DataSciencester 的网络中，Thor（`id` 为 4）只有两个联系数，Dunn（`id` 为 1）有三个。从网络关系图中看，直观上感觉 Thor 处于中心地位。第 21 章将考察网络关系的更多细节，探讨更多关于中心性的复杂概念，它们可能与直觉一致，也可能不一致。

1.3.2 你可能知道的数据科学家

当你正在填写新员工入职表格时，人力部门的副总来到你桌旁。她想鼓励数据科学家们多多联系，因而希望你设计一个“你可能知道的数据科学家”的提示函数。

你的直觉是用户可能会认识朋友的朋友。这不难计算：对某个用户，依次计算每个朋友的朋友，最后合并结果：

```
def friends_of_friend_ids_bad(user):
    # foaf是“朋友的朋友”的英文简写
    return [foaf["id"]
            for friend in user["friends"]    # 对每一位用户的朋友
            for foaf in friend["friends"]]   # 得到他们的朋友
```

当我们对 `users[0]`（Hero）调用上面这个函数时，它的结果显示为：

```
[0, 2, 3, 0, 1, 3]
```

因为 Hero 是他两位朋友的朋友，所以结果中包含两次用户 0。同时，因为用户 1 和用户 2 都是 Hero 的朋友，所以也包含在结果中。此外，由于用户 Chi 可以通过用户 1 和用户 2 与用户 0 联系，所以包含了他两次：

```
print [friend["id"] for friend in users[0]["friends"]]  # [1, 2]
print [friend["id"] for friend in users[1]["friends"]]  # [0, 2, 3]
print [friend["id"] for friend in users[2]["friends"]]  # [0, 1, 3]
```

有趣的是，人们可以通过朋友的朋友相互认识。受此启发，我们也许可以设计一个共同的朋友，由他来表示朋友的计数。同时，为了排除那些已经成为朋友的用户，我们需要设计一个辅助函数来实现这个功

能：

```
from collections import Counter                                   # 默认未加载

def not_the_same(user, other_user):
    """two users are not the same if they have different ids"""
    return user["id"] != other_user["id"]

def not_friends(user, other_user):
    """other_user is not a friend if he's not in user["friends"];
    that is, if he's not_the_same as all the people in user["friends"]"""
    return all(not_the_same(friend, other_user)
               for friend in user["friends"])

def friends_of_friend_ids(user):
    return Counter(foaf["id"]
                   for friend in user["friends"]    # 对我的每一位朋友
                   for foaf in friend["friends"]    # 计数他们的朋友
                   if not_the_same(user, foaf)      # 既不是我
                   and not_friends(user, foaf))     # 也不是我的朋友

print friends_of_friend_ids(users[3])               # Counter({0: 2, 5: 1})
```

这个输出结果正确无误地说明了 Chi（id 为 3）和 Hero（id 为 0）有两个共同的朋友，和 Clive（id 为 5）有一个共同的朋友。

出于一个数据科学家的直觉，你可能会喜欢结交有共同兴趣的人（这个例子很好地展示了数据科学家的专业技能）。咨询之后，你设计出如下列表，每个元素都是成对数据 (user_id, interest)：

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
    (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
    (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
    (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python"),
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"), (4, "decision trees"),
    (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
    (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
]
```

例如，Thor（id 为 4）与 Devin（id 为 7）没有共同的朋友，但他们对机器学习都感兴趣。

如果需要找出对某种事物有共同爱好的用户，很容易设计出相应的函数：

```
def data_scientists_who_like(target_interest):
    return [user_id
            for user_id, user_interest in interests
            if user_interest == target_interest]
```

但是，上面的算法每次搜索都需要遍历整个兴趣列表，如果用户很多或者用户的兴趣很多（或我们只是想多进行一些查找），这种算法的时间和空间成本会很大，因此最好能建立一个从兴趣到用户的索引直接搜索：

```
from collections import defaultdict

# 键是interest，值是带有这个interest的user_id的列表
user_ids_by_interest = defaultdict(list)

for user_id, interest in interests:
    user_ids_by_interest[interest].append(user_id)
```

以及另一个从用户到兴趣的索引：

```
# 键是user_id，值是对那些user_id的interest的列表
interests_by_user_id = defaultdict(list)
for user_id, interest in interests:
    interests_by_user_id[user_id].append(interest)
```

现在，给定一个用户，可以方便地找到与他共同爱好最多的用户：

- 迭代这个用户的兴趣；
- 针对这个用户的每一种兴趣，寻找这种兴趣的其他用户，并迭代；
- 记录每一个用户在循环中出现的次数。

```
def most_common_interests_with(user):
    return Counter(interested_user_id
```

```
        for interest in interests_by_user_id[user["id"]]
        for interested_user_id in user_ids_by_interest[interest]
        if interested_user_id != user["id"])
```

接下来，结合共同的朋友和共同的兴趣，可以建立一个更丰富的功能“你应该知道的数据科学家”，这类应用我们将在第 22 章探讨。

1.3.3　工资与工作年限

当你正准备去吃午饭时，公共关系部门的副总突然来到你的办公室，问你能不能给他提供一些关于数据科学家的收入的有趣数据。当然，大家对工资数据都很敏感，他想办法搞出了一份匿名文件，其中包含每位用户的工资（`salary`）和作为数据科学家的工作年限（`tenure`）：

```
salaries_and_tenures = [(83000, 8.7), (88000, 8.1),
                        (48000, 0.7), (76000, 6),
                        (69000, 6.5), (76000, 7.5),
                        (60000, 2.5), (83000, 10),
                        (48000, 1.9), (63000, 4.2)]
```

很自然地，第一步是绘制数据（第 3 章有详细介绍），如图 1-3 所示。

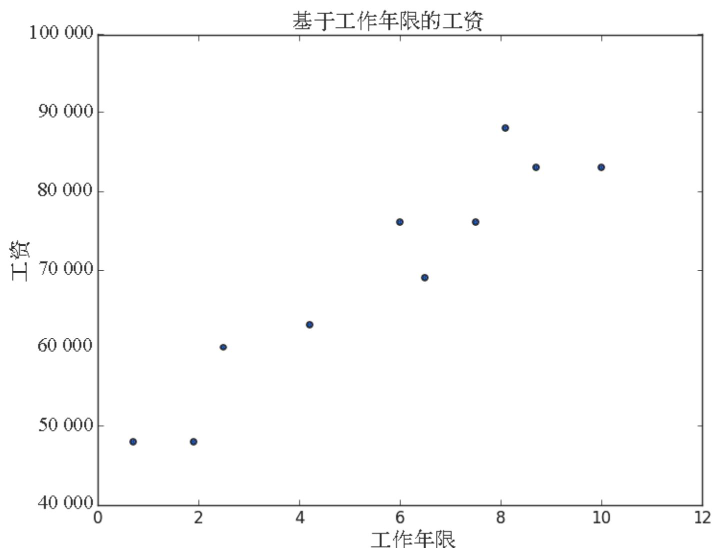


图 **1-3**：基于工作年限的工资图

图中所示的关系很明显，工作年限越长的人收入越高。接下来需要考虑的是如何将这个结果转化成更有趣的事实。你首先想到了考察一下大家的年均收入：

```
# 键是year，值是对每一个tenure的salary的列表
salary_by_tenure = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    salary_by_tenure[tenure].append(salary)

# 键是year，每个值是相应tenure的平均salary
average_salary_by_tenure = {
    tenure : sum(salaries) / len(salaries)
    for tenure, salaries in salary_by_tenure.items()
}
```

实际上，任何两个用户都没有相同的工作年限，所以上述计算结果作用有限，它仅仅说明了每个用户独立的收入：

```
{0.7: 48000.0,
 1.9: 48000.0,
 2.5: 60000.0,
 4.2: 63000.0,
 6: 76000.0,
 6.5: 69000.0,
 7.5: 76000.0,
 8.1: 88000.0,
 8.7: 83000.0,
 10: 83000.0}
```

一个更有意义的计算方式是把用户的工作年限分组：

```
def tenure_bucket(tenure):
    if tenure < 2:
        return "less than two"
    elif tenure < 5:
        return "between two and five"
    else:
        return "more than five"
```

再将每组的工资合并：

```
# 键是tenure bucket，值是相应bucket的salary的列表
salary_by_tenure_bucket = defaultdict(list)

for salary, tenure in salaries_and_tenures:
    bucket = tenure_bucket(tenure)
    salary_by_tenure_bucket[bucket].append(salary)
```

最后，计算每个分组的平均工资：

```
# 键是tenure bucket，值是对那个bucket的average salary
average_salary_by_bucket = {
  tenure_bucket : sum(salaries) / len(salaries)
  for tenure_bucket, salaries in salary_by_tenure_bucket.iteritems()
}
```

这是更有趣的结果：

```
{'between two and five': 61500.0,
 'less than two': 48000.0,
 'more than five': 79166.66666666667}
```

现在，你得到结论："有 5 年以上工作年限的数据科学家比同行新人的收入高 65%。"

但是，我们的选择是任意的。我们原本希望说明的是，平均看来，更多的工作年限意味着更多的工资收入。为了得到更多有趣的结论，我们可以预测一些未知年限的工资。我们将在第 14 章探讨这个想法。

1.3.4　付费账户

当你回到桌旁时，收益部门的副总在等你。他想更好地了解哪些用户会为账户付费，哪些用户不会。（他知道他们的名字，但这些信息不是特别有用。）

你注意到在工作年限和付费账户之间似乎存在一种对应关系：

```
0.7 paid
1.9 unpaid
2.5 paid
4.2 unpaid
6   unpaid
6.5 unpaid
7.5 unpaid
```

```
8.1 unpaid
8.7 paid
10  paid
```

那些新手和资历很深的用户倾向于付费，而那些具有中等工作年限的用户则倾向于不付费。

由此，如果你打算创建一个模型——尽管这点数据对创建模型肯定是不够的——你会试图对新手和资深用户预测“付费”，而对具有中等工作年限的用户预测“不付费”：

```
def predict_paid_or_unpaid(years_experience):
  if years_experience < 3.0:
    return "paid"
  elif years_experience < 8.5:
    return "unpaid"
  else:
    return "paid"
```

当然，我们会完全紧盯这个切入口。

利用更多的数据（和更多的数学计算），我们可以基于用户的工作年限来预测用户付费的可能性。我们会在第 16 章研究这类问题。

1.3.5　兴趣主题

当你正准备结束第一天的工作时，内容策略部门的副总来向你要数据，想了解什么样的主题更令用户感兴趣，以便据此规划他的博客日历。你已经有了来自友邻推荐项目的原始数据：

```
interests = [
    (0, "Hadoop"), (0, "Big Data"), (0, "HBase"), (0, "Java"),
    (0, "Spark"), (0, "Storm"), (0, "Cassandra"),
    (1, "NoSQL"), (1, "MongoDB"), (1, "Cassandra"), (1, "HBase"),
    (1, "Postgres"), (2, "Python"), (2, "scikit-learn"), (2, "scipy"),
    (2, "numpy"), (2, "statsmodels"), (2, "pandas"), (3, "R"), (3, "Python
    (3, "statistics"), (3, "regression"), (3, "probability"),
    (4, "machine learning"), (4, "regression"), (4, "decision trees"),
    (4, "libsvm"), (5, "Python"), (5, "R"), (5, "Java"), (5, "C++"),
    (5, "Haskell"), (5, "programming languages"), (6, "statistics"),
    (6, "probability"), (6, "mathematics"), (6, "theory"),
    (7, "machine learning"), (7, "scikit-learn"), (7, "Mahout"),
    (7, "neural networks"), (8, "neural networks"), (8, "deep learning"),
    (8, "Big Data"), (8, "artificial intelligence"), (9, "Hadoop"),
    (9, "Java"), (9, "MapReduce"), (9, "Big Data")
```

```
]
```

一种简单（但并不激动人心）的方法是仅仅数一下兴趣词汇的个数：

1. 小写每一种兴趣（因为不同的用户不一定会大写他们的兴趣）；

2. 把它划分为单词；

3. 数一数结果。

用下面的代码：

```
words_and_counts = Counter(word
                           for user, interest in interests
                           for word in interest.lower().split())
```

列出出现一次以上的词汇是很容易的：

```
for word, count in words_and_counts.most_common():
    if count > 1:
        print word, count
```

这给出了你所期待的结果（除非你想让“scikit-learn”分化成两个词，那样就不会得到预期的结果了）：

```
learning 3
java 3
python 3
big 3
data 3
hbase 2
regression 2
cassandra 2
statistics 2
probability 2
hadoop 2
networks 2
machine 2
neural 2
scikit-learn 2
r 2
```

我们会在第 20 章学习更多从数据中提取主题的复杂方法。

1.4 展望

第一天的工作非常成功！你肯定很累，赶快趁没人继续问问题时回家吧。明天是新员工培训，所以今晚好好休息一下。（当然啦，你还没接受培训就已经工作一整天了！明天去找 HR 吧。）

第 2 章　Python 速成

难以置信，**25** 年以来 **Python** 始终广受追捧。

——迈克尔 · 佩林

DataSciencester 要求所有新员工接受入职培训，其中最有趣的部分当属 Python 速成。

本章不是完整的 Python 培训教程，而仅强调其中对我们最重要的部分（其中有些部分并非 Python 培训教程的重点）。

2.1　基础内容

2.1.1　Python获取

可以从 python.org（https://www.python.org/）网站下载 Python。但是对于没有安装过 Python 的读者，特别推荐安装 Anaconda 版本（https://www.continuum.io/downloads），这个版本涵盖了数据科学工作需要用到的大多数库。

当我写本书时，Python 的最新版本是 3.4。但是，在 DataSciencester，我们使用一个旧的可靠版本，Python 2.7。Python 3 无法向后兼容 Python 2，因此很多重要功能只能在 Python 2.7 上良好运行。数据科学社区中，2.7 版本一直是主流，我们也和它保持一致。请设法安装这个版本。

如果你没有 Anaconda 版本，那么需要安装 pip（https://pypi.python.org/pypi/pip），这是一个 Python 包管理器，可以用来方便地安装第三方包（其中有些是我们必需的）。IPython（http://ipython.org/）也不错，操作界面更友好。

（如果你已经安装了 Anaconda 版本，它会很好地和 pip 与 IPython 兼容。）运行：

```
pip install ipython
```

如果碰到难解的错误信息，可以在网上搜索解决办法。

2.1.2 Python之禅

Python 的设计原则（http://legacy.python.org/dev/peps/pep-0020/）有着禅宗的意味，你输入 **import this** 就能在 Python 解释器中一窥玄机。

讨论最多的原则之一是：

> 应该有一个——且最好只有一个——明显的方式去完成工作。

按照这种“明显”的方式（对一个新人来说可能根本不明显）编写的代码常常被称为具有“Python 风格”。尽管本书不是专门介绍 Python 的书，但我们时不时地会比较 Python 风格和非 Python 风格的方式在解决相同问题时的差异，而且你常常会发现 Python 风格是更好的解决方式。

2.1.3 空白形式

许多编程语言用大括号分隔代码块。Python 使用缩进的方式：

```
for i in [1, 2, 3, 4, 5]:
    print i                         # "for i"程序块的第一行
    for j in [1, 2, 3, 4, 5]:
        print j                     # "for j"程序块的第一行
        print i + j                 # "for j"程序块的最后一行
    print i                         # "for i"程序块的最后一行
print "done looping"
```

这样会使 Python 代码非常易读，但这也意味着你需要非常小心格式。系统会省略方括号和圆括号中的空白，这对冗长的计算非常有用：

```
long_winded_computation = (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 1
```

并能使代码更易读：

```
list_of_lists = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

```
easier_to_read_list_of_lists = [ [1, 2, 3],
                                 [4, 5, 6],
                                 [7, 8, 9] ]
```

你可以用反斜线代表一个语句在下一行续写，尽管这个例子并不常见：

```
two_plus_three = 2 + \
                 3
```

空白形式的一个后果是很难将代码复制并粘贴到 Python 的 shell。比如，如果你粘贴代码：

```
for i in [1, 2, 3, 4, 5]:

    # 注意这个空行
    print i
```

在粘贴入一般的 Python shell 后，会得到以下提示：

```
IndentationError: expected an indented block
```

因为解释器认为空行表示 `for` 循环的终结。

IPython 有一个奇妙的函数 `%paste`，可以正确地复制剪贴板上的内容，包括空白在内。这个函数足以成为选择使用 IPython 的好理由。

2.1.4　模块

Python 的某些特征默认不加载，包含了语言本身的部分特征，也包含了需读者自行下载的第三方特征。为了使用这些特征，你需要导入包含这些特征的模块。

一种方式是简单地导入模块本身：

```
import re
my_regex = re.compile("[0-9]+", re.I)
```

这里的 `re` 代表包含了处理正则表达式需要的函数与常量的模块。输

入 `import` 之后，你可以通过加前缀 `re.` 来直接调用模块中的函数。

如果你的代码中已经有了不同的 `re`，可以使用别名：

```
import re as regex
my_regex = regex.compile("[0-9]+", regex.I)
```

如果模块名冗长，或者你需要敲很多字符，那不妨试试这个方法。比如，你想对数据用模块 `matplotlib` 来进行可视化，标准转换如下：

```
import matplotlib.pyplot as plt
```

如果你需要一个模块中的一些特定值，可以显式导入，直接使用，不必提前获取权限，如下所示：

```
from collections import defaultdict, Counter
lookup = defaultdict(int)
my_counter = Counter()
```

如果你是破坏者，可以将模块的全部内容导入命名空间，这也许会不加提示地覆盖原先定义的变量：

```
match = 10
from re import *       # 呃，re有一个match函数
print match            # "<function re.match>"
```

但是，不是破坏者就别这样做。

2.1.5　算法

Python 2.7 默认整除，因此 `5 / 2` 等于 `2`。但是这并不总是我们想要的，因此文件常常需要这样开始：

```
from __future__ import division
```

这样 `5 / 2` 等于 `2.5`。本书中的所有示例程序采用新除法。在少数

需要整除的情形下，我们可以通过双斜线 `5 // 2` 表示。

2.1.6　函数

函数是一种规则，输入零或者其他数，得到相应的输出。在 Python 中，我们用 `def` 定义函数：

```
def double(x):
    """this is where you put an optional docstring
    that explains what the function does.
    for example, this function multiplies its input by 2"""
    return x * 2
```

Python 函数是第一类函数，第一类函数意味着可以将它们赋给其他变量，也可以像其他参数一样传递给函数：

```
def apply_to_one(f):
    """calls the function f with 1 as its argument"""
    return f(1)

my_double = double              # 指向之前定义的函数
x = apply_to_one(my_double)     # 等于2
```

也很容易生成简短的匿名函数，或者 lambda：

```
y = apply_to_one(lambda x: x + 4)      # 等于5
```

你可以将 lambda 赋给变量，尽管大部分人会建议你用 `def`：

```
another_double = lambda x: 2 * x       # 别这么做
def another_double(x): return 2 * x    # 要这么做
```

默认参数也可以赋值给函数参数，当你需要默认值以外的值时需要具体说明：

```
def my_print(message="my default message"):
    print message
my_print("hello")         # 打印"hello"
my_print()                # 打印"my default message"
```

有时候通过名字指定参数会有用：

```
def subtract(a=0, b=0):
    return a - b

subtract(10, 5) # 返回5
subtract(0, 5)  # 返回-5
subtract(b=5)   # 和前一句一样
```

我们可以生成很多很多函数。

2.1.7　字符串

字符串可以用单引号或者双引号标注分隔出来（但引号需要配对，即单引号对单引号，双引号对双引号）：

```
single_quoted_string = 'data science'
double_quoted_string = "data science"
```

Python 用反斜线来为特殊字符编码。比如：

```
tab_string = "\t"     # 表示tab字符
len(tab_string)       # 是1
```

如果你希望反斜线仅仅代表反斜线本身（你在 Windows 系统中的文件夹或者正则表达式中也许会遇到），那么可以使用命令 `r""` 生成一个原始的字符串：

```
not_tab_string = r"\t"     # 表示字符'\'和't'
len(not_tab_string)        # 是2
```

你可以通过三重［两重］引号来生成多行的字符串：

```
multi_line_string = """This is the first line.
and this is the second line
and this is the third line"""
```

2.1.8　异常

当发生意外时，Python 会报出异常。如果不处理，异常会引起程序崩溃。你可以用 `try` 和 `except` 来解决：

```
try:
    print 0 / 0
except ZeroDivisionError:
    print "cannot divide by zero"
```

在许多语言中，异常意味着坏情形。但在 Python 中，你不必害怕遇见异常，只要能写出清晰的代码，我们时而会这样做。

2.1.9 列表

Python 中最基本的数据结构是列表（`list`）。一个列表是一个有序的集合。（这和其他语言中的数组概念类似，但增加了函数功能。）

```
integer_list = [1, 2, 3]
heterogeneous_list = ["string", 0.1, True]
list_of_lists = [ integer_list, heterogeneous_list, [] ]

list_length = len(integer_list)      # 等于3
list_sum    = sum(integer_list)      # 等于6
```

你可以通过方括号对列表的第 n 个元素读值和赋值：

```
x = range(10)    # 是列表[0, 1, ..., 9]
zero = x[0]      # 等于0，列表是0-索引的
one = x[1]       # 等于1
nine = x[-1]     # 等于9，最后一个元素的Python惯用法
eight = x[-2]    # 等于8，倒数第二个元素的Pyhton惯用法
x[0] = -1        # 现在x是[-1, 1, 2, 3, ..., 9]
```

你也可以用方括号来“切取”列表：

```
first_three   = x[:3]                # [-1, 1, 2]
three_to_end = x[3:]                 # [3, 4, ..., 9]
one_to_four = x[1:5]                 # [1, 2, 3, 4]
last_three = x[-3:]                  # [7, 8, 9]
without_first_and_last = x[1:-1]     # [1, 2, ..., 8]
copy_of_x = x[:]                     # [-1, 1, 2, ..., 9]
```

Python 可以通过操作符 `in` 确认列表成员：

```
1 in [1, 2, 3]     # True
0 in [1, 2, 3]     # False
```

确认每次都会遍历列表元素。这意味着除非列表很小，否则就不应该进行确认（除非你不在乎确认需要花费多长时间）。

将列表串连起来很容易：

```
x = [1, 2, 3]
x.extend([4, 5, 6])    # x现在是[1,2,3,4,5,6]
```

如果不希望改变原序列 x，你可以使用列表加法：

```
x = [1, 2, 3]
y = x + [4, 5, 6]     # y是[1, 2, 3, 4, 5, 6]；x是不变的
```

更常见的做法是，一次在原列表上只增加一项：

```
x = [1, 2, 3]
x.append(0)         # x现在是[1, 2, 3, 0]
y = x[-1]           # 等于0
z = len(x)          # 等于4
```

如果你知道列表中元素的个数，可以方便地从中提取值：

```
x, y = [1, 2]     # 现在x是1，y是2
```

不过，如果等式两端元素个数不同，会报出提示 ValueError。

如果你希望忽略某些值，常见的选择是使用下划线：

```
_, y = [1, 2]     # 现在y==2，不用关心第一个元素是什么
```

2.1.10　元组

元组是列表的亲表哥。你对列表做的很多操作都可以对元组做，但不包括修改。元组通过圆括号（或者什么都不加）而不是方括号来给出

具体的描述：

```
my_list = [1, 2]
my_tuple = (1, 2)
other_tuple = 3, 4
my_list[1] = 3      # my_list现在是[1, 3]

try:
    my_tuple[1] = 3
except TypeError:
    print "cannot modify a tuple"
```

元组是通过函数返回多重值的便捷方法：

```
def sum_and_product(x, y):
    return (x + y),(x * y)

sp = sum_and_product(2, 3)     # 等于(5，6)
s, p = sum_and_product(5, 10)  # s是15，p是50
```

元组（和列表）都可以进行多重赋值（multiple assignment）：

```
x, y = 1, 2      # 现在x是1，y是2
x, y = y, x      # Python风格的互换变量，现在x是2，y是1
```

2.1.11　字典

Python 中的另一种基本数据结构是字典，它将值与键联系起来，让我们可以通过键快速找到对应值：

```
empty_dict = {}                          # Python风格
empty_dict2 = dict()                     # 更少的Python风格
grades = { "Joel" : 80, "Tim" : 95 }     # 字典
```

你也可以通过方括号查找键的值：

```
joels_grade = grades["Joel"]             # 等于80
```

如果你找的键不在字典中，会得到 `KeyError` 报错：

```
try:
    kates_grade = grades["Kate"]
except KeyError:
    print "no grade for Kate!"
```

你可以用 in 确认键的存在：

```
joel_has_grade = "Joel" in grades     # 正确
kate_has_grade = "Kate" in grades     # 错误
```

如果查找的键在字典中不存在，字典可以通过方法 get 返回默认值（而非报出异常）：

```
joels_grade = grades.get("Joel", 0)    # 等于80
kates_grade = grades.get("Kate", 0)    # 等于0
no_ones_grade = grades.get("No One")   # 默认的默认值为None
```

你可以通过方括号来为键值对赋值：

```
grades["Tim"] = 99                     # 替换了旧的值
grades["Kate"] = 100                   # 增加了第三个记录
num_students = len(grades)             # 等于3
```

我们常常使用字典作为代表结构数据的简单方式：

```
tweet = {
    "user" : "joelgrus",
    "text" : "Data Science is Awesome",
    "retweet_count" : 100,
    "hashtags" : ["#data", "#science", "#datascience", "#awesome", "#yolo"
}
```

除了查找特定的键，我们还可以查找所有值：

```
tweet_keys   = tweet.keys()      # 键的列表
tweet_values = tweet.values()    # 值的列表
tweet_items  = tweet.items()     # （键， 值）元组的列表

"user" in tweet_keys             # True，使用慢速的列表
"user" in tweet                  # 更符合Python惯用法，使用快速的字典
"joelgrus" in tweet_values       # True
```

字典的键不可改变，尤其是不能将列表作为键。如果你需要一个多维的键，应该使用元组或设法把键转换成字符串。

1. `defaultdict`

假设你需要计算某份文件中的单词数目。一个明显的方式是，建立一个键是单词、值是单词出现次数的字典。每次你查到一个单词，如果字典中存在这个词，就在该词的计数上增加 1，如果字典中没有这个词，就把这个词增加到这个字典中：

```
word_counts = {}
for word in document:
    if word in word_counts:
        word_counts[word] += 1
    else:
        word_counts[word] = 1
```

当查找缺失值碰到异常报出时，你可以遵循“与其瞻前顾后，不如果断行动”（Forgiveness is better than permission）的原则，果断处理异常：

```
word_counts = {}
for word in document:
    try:
        word_counts[word] += 1
    except KeyError:
        word_counts[word] = 1
```

第三种方法是使用 `get`，这种处理缺失值的方法比较优雅：

```
word_counts = {}
for word in document:
    previous_count = word_counts.get(word, 0)
    word_counts[word] = previous_count + 1
```

以上三种方法都略显笨拙，这是 `defaultdict` 的意义之所在。一个 `defaultdict` 相当于一个标准的字典，除了当你查找一个没有包含在内的键时，它用一个你提供的零参数函数建立一个新的键，并为它的值增加 1。为了使用 `defaultdict`，你需要将其从集合中导出：

```
from collections import defaultdict
```

```
word_counts = defaultdict(int)                # int()生成0

for word in document:
    word_counts[word] += 1
```

这对列表、字典或者你自己的函数都有用：

```
dd_list = defaultdict(list)                   # list()生成一个空列表
dd_list[2].append(1)                          # 现在dd_list包含{2:[1]}

dd_dict = defaultdict(dict)                   # dict()产生一个新字典
dd_dict["Joel"]["City"] = "Seattle"           # { "Joel" : { "City" : Seattle"}}

dd_pair = defaultdict(lambda: [0, 0])
dd_pair[2][1] = 1                             # 现在dd_pair包含{2: [0,1]}
```

当我们用字典“收集”某些键对应的结果，并且不希望每次查找某键是否存在都遍历一遍的时候，defaultdict 非常有用。

2. Counter

一个计数器将一个序列的值转化成一个类似于整型的标准字典（即 defaultdict(int)）的键到计数的对象映射。我们主要用它来生成直方图：

```
from collections import Counter
c = Counter([0, 1, 2, 0])          # c是（基本的）{ 0 : 2, 1 : 1, 2 : 1 }
```

这给我们提供了一个用来解决单词计数问题的很简便的方法：

```
word_counts = Counter(document)
```

一个 Counter 实例带有的 most_common 方法的例子如下：

```
# 打印10个最常见的词和它们的计数
for word, count in word_counts.most_common(10):
    print word, count
```

2.1.12　集合

另一种数据结构是集合（set），它表示为一组不同的元素：

```
s = set()
s.add(1)          # s现在是1
s.add(2)          # s现在是{1,2}
s.add(2)          # s还是{1,2}
x = len(s)        # 等于2
y = 2 in s        # 等于True
z = 3 in s        # 等于False
```

我们使用集合的原因主要有两个。第一个是集合上有一种非常快速的操作：in。如果我们有大量的项目，希望对它的成分进行测试，那么使用集合比使用列表要合适得多：

```
stopwords_list = ["a","an","at"] + hundreds_of_other_words + ["yet", "you"
"zip" in stopwords_list     # False，但需要检查每个元素

stopwords_set = set(stopwords_list)
"zip" in stopwords_set      # 非常快地检查
```

第二个原因是便于在一个汇总中寻找其中离散的项目：

```
item_list = [1, 2, 3, 1, 2, 3]
num_items = len(item_list)              # 6
item_set = set(item_list)               # {1, 2, 3}
num_distinct_items = len(item_set)      # 3
distinct_item_list = list(item_set)     # [1, 2, 3]
```

我们使用 set 的频率要远低于 dict 和 list。

2.1.13　控制流

和大多数编程语言一样，你可以用 if 语句来执行一种有条件的行动：

```
if 1 > 2:
    message = "if only 1 were greater than two..."
elif 1 > 3:
    message = "elif stands for 'else if'"
else:
    message = "when all else fails use else (if you want to)"
```

也可以在一行语句中使用 if-then-else，我们有时候需要这么做：

```
parity = "even" if x % 2 == 0 else "odd"
```

Python 也有 while 循环：

```
x = 0
while x < 10:
    print x, "is less than 10"
    x += 1
```

尽管我们更常用 for 和 in：

```
for x in range(10):
    print x, "is less than 10"
```

如果你需要更复杂的逻辑表达式，可以使用 continue 和 break：

```
for x in range(10):
    if x == 3:
        continue          # 直接进入下次迭代
    if x == 5:
        break             # 完全退出循环
    print x
```

上面的语句会打印 0、1、2 和 4。

2.1.14 真和假

Python 的布尔数除了首字母大写之外，其他用法和大多数别的语言类似：

```
one_is_less_than_two = 1 < 2          # 等于True
true_equals_false = True == False     # 等于False
```

Python 使用 None 来表示一个不存在的值，它类似别的语言中的 null：

```
x = None
```

```
print x == None     # 打印True，但这并非Python的惯用法
print x is None     # 打印True，符合Python的惯用法
```

Python 可以使用任何可被认为是布尔数的值。下面这些都是“假”（Falsy）：

- False
- None
- [ ]（一个空 list）
- { }（一个空 dict）
- ""
- set()
- 0
- 0.0

还有很多值可作为真（True）来处理。这样你可以很容易地使用 if 语句来对空列表、空字符串或空字典等进行测试。有时候，如果你没有意识到这种行为，会引入一些微妙的 bug：

```
s = some_function_that_returns_a_string()
if s:
    first_char = s[0]
else:
    first_char = ""
```

另一种简单的方法是：

```
first_char = s and s[0]
```

这是因为第一个值为“真”时，and 运算符返回它的第二个值，否则返回第一个值。类似地，如果 x 的取值可能是一个数也可能是 None，那么以下代码的结果就必然会是一个数字：

```
safe_x = x or 0
```

```

```

Python 还有一个 `all` 函数，它的取值是一个列表，当列表的每个元素都为真时返回 `True`。

Python 还有一个 `any` 函数，当取值的列表中至少有一个元素为真时，它返回 `True`：

```
all([True, 1, { 3 }])     # True
all([True, 1, {}])        # False，{}为假
any([True, 1, {}])        # True
all([])                   # True，列表里没有假的元素
any([])                   # False，列表里没有真的元素
```

2.2　进阶内容

现在我们来看一些比较高级的 Python 特性，这些特性对开展数据工作特别有用。

2.2.1　排序

每个 Python 列表都有一个 `sort` 方法可以恰当地排序。如果你不想弄乱你的列表，可以使用 `sorted` 函数，它会返回一个新列表：

```
x = [4,1,2,3]
y = sorted(x)          # 结果是[1,2,3,4]，但x没有变
x.sort()               # x变为[1,2,3,4]
```

默认情况下，`sort`（和 `sorted`）基于元素之间的朴素比较从最小值到最大值对列表进行排序。

如果你想把元素按从最大值到最小值进行排序，可以指定参数 `reverse=True`。除了比较元素本身，你还可以通过指定键来对函数的结果进行比较：

```
# 通过绝对值对列表元素从最大到最小排序
x = sorted([-4,1,-2,3], key=abs, reverse=True) # 是[-4,3,-2,1]

# 从最高数到最低数排序单词和计数
wc = sorted(word_counts.items(),
            key=lambda (word, count): count,
```

```
                 reverse=True)
```

2.2.2 列表解析

我们有时可能会想把一个列表转换为另一个列表，例如只保留其中一些元素，或更改其中一些元素，或者同时做这两种变动。可以执行这种操作的 Python 技巧叫作列表解析（list comprehension）：

```
even_numbers = [x for x in range(5) if x % 2 == 0]     # [0, 2, 4]
squares      = [x * x for x in range(5)]               # [0, 1, 4, 9, 16]
even_squares = [x * x for x in even_numbers]           # [0, 4, 16]
```

类似地，你也可以把列表转换为字典或集合：

```
square_dict = { x : x * x for x in range(5) }     # { 0:0, 1:1, 2:4, 3:9, 4
square_set  = { x * x for x in [1, -1] }          # { 1 }
```

如果你不需要来自原列表中的值，常规的方式是使用下划线作为变量：

```
zeroes = [0 for _ in even_numbers]      # 和even_numbers有相同的长度
```

列表解析可以包括多个 `for` 语句：

```
pairs = [(x, y)
         for x in range(10)
         for y in range(10)]     # 100个对(0,0) (0,1) ... (9,8), (9,9)
```

其中后面的 `for` 语句可以使用前面的 `for` 语句的结果：

```
increasing_pairs = [(x, y)                       # 只考虑x < y的对
                    for x in range(10)           # range(lo, hi) 与之相等
                    for y in range(x + 1, 10)]   # [lo, lo + 1, ..., hi -
```

我们会经常用到列表解析。

2.2.3 生成器和迭代器

列表的一个问题是它很容易变得非常大。range(1000000) 能创建一个有 100 万个元素的列表：如果你需要每次只处理其中一个元素，这将会是极大的资源浪费（或会导致内存不足）；如果你只需要前面的几个值，那对整个列表都进行计算也是一种浪费。

生成器（generator）是一种可以对其进行迭代（对我们来说，通常使用 for 语句）的程序，但是它的值只按需延迟（lazily）产生。

创建生成器的一种方法是使用函数和 yield 运算符：

```
def lazy_range(n):
    """a lazy version of range"""
    i = 0
    while i < n:
        yield i
        i += 1
```

下面的循环会每次消耗一个 yield 值直到一个也不剩：

```
for i in lazy_range(10):
    do_something_with(i)
```

（Python 确实有一个和 lazy_range 一样的函数，叫作 xrange，并且在 Python 3 中，range 函数本身就是延迟的。）这意味着，你甚至可以创建一个无限的序列：

```
def natural_numbers():
    """returns 1, 2, 3, ..."""
    n = 1
    while True:
        yield n
        n += 1
```

尽管在没有使用某种 break 逻辑语句时，你不应该做这种迭代的。



延迟的缺点是，你只能通过生成器迭代一次。如果需要多次迭代某个对象，你就需要每次都重新创建一个生成器，或者使用列表。

第二种创建生成器的方法是使用包含在圆括号中的 for 语句解析：

```
lazy_evens_below_20 = (i for i in lazy_range(20) if i % 2 == 0)
```

前面提过，每个 dict 都有一个 items() 方法可以返回它的键值对的列表。更常见的做法是使用 iteritems() 方法：当我们在列表上迭代的时候它延迟 yield 为每次一个键值对。

2.2.4 随机性

当我们学习数据科学时，会经常需要生成随机数。可以使用 random 模块生成随机数：

```
import random

four_uniform_randoms = [random.random() for _ in range(4)]

# [0.8444218515250481,
#  0.7579544029403025,
#  0.420571580830845,
#  0.25891675029296335]
# random.random()生成在0-1之间均匀分布的随机数，是最常用的随机函数
```

random 模块实际上生成的是基于一种内部状态的确定性的伪随机数。如果你想得到可复生的结果，可以用 random.seed 生成随机数种子：

```
random.seed(10)             # 设置随机数种子为10
print random.random()       # 0.57140259469
random.seed(10)             # 重设随机数种子为10
print random.random()       # 再次得到0.57140259469
```

有时候我们用 random.randrange 生成随机数，它会取 1 到 2 个参数，并从对应的 range() 函数随机选择一个元素返回：

```
random.randrange(10)      # 从range(10) = [0, 1, ..., 9]中随机选取
random.randrange(3, 6)    # 从range(3, 6) = [3, 4, 5]中随机选取
```

还有其他一些比较方便的方法，如 Random.shuffle 可随机地重排列表中的元素：

```
up_to_ten = range(10)
random.shuffle(up_to_ten)
```

```
print up_to_ten
# [2, 5, 1, 9, 7, 3, 8, 6, 4, 0]，（你的结果可能不同）
```

如果你需要从列表中随机取一个元素，可以使用 random.choice：

```
my_best_friend = random.choice(["Alice", "Bob", "Charlie"])     # 对我来说是
```

如果你需要不替换地（即不重复地）随机选择一个元素的样本，可以使用 random.sample：

```
lottery_numbers = range(60)
winning_numbers = random.sample(lottery_numbers, 6)  # [16, 36, 10, 6, 25,
```

选择一个允许替换的（即允许重复的）元素样本，只需多次调用 random.choice 即可：

```
four_with_replacement = [random.choice(range(10))
                         for _ in range(4)]
# [9, 4, 4, 2]
```

2.2.5　正则表达式

正则表达式提供了一种搜索文本的方法。它超乎想象地有用，但同时也相当复杂，以至于需要专门的书籍来讲解。之后的内容会频繁涉及正则表达式，届时我们再详述，这里只给出 Python 中如何使用正则表达式的例子：

```
import re

print all([                                 # 所有这些语句都为true，因为
    not re.match("a", "cat"),               # * 'cat'不以'a'开头
    re.search("a", "cat"),                  # * 'cat'里有一个字符'a'
    not re.search("c", "dog"),              # * 'dog'里没有字符'c'
    3 == len(re.split("[ab]", "carbs")),    # * 分割掉a,b，剩余长度为3
    "R-D-" == re.sub("[0-9]", "-", "R2D2")  # 用虚线进行位的替换
    ] # 打印True
```

2.2.6　面向对象的编程

就像许多语言一样，Python 允许你定义类（class）。类可以封装对象和函数来对它们进行操作。有时候我们会用类来使代码更加干净整洁。解释类的用法的最简单方式可能是构建一个有超多注释的例子。

假设没有内置的 Python 集合，那我们可能会想到去创建自己的 `Set` 类。

我们创建的类会有什么样的行为呢？给定一个 `Set`，我们需要能在其中加入（`add`）项目，移除（`remove`）项目，以及检查其中是否包含（`contains`）某个值。我们把这些功能创建为成员（member）函数，意思是我们可以通过在 `Set` 对象后面加点（`.`）来访问它们：

```
# 按惯例，我们给下面的类起个PascalCase的名字
class Set:

    # 这些是成员函数
    # 每个函数都取第一个参数"self"（另一种惯例）
    # 它表示所用到的特别的集合对象

    def __init__(self, values=None):
        """This is the constructor.
        It gets called when you create a new Set.
        You would use it like
        s1 = Set()          # 空集合
        s2 = Set([1,2,2,3])  # 用值初始化"""

        self.dict = {} # Set的每一个实例都有自己的dict属性
                       # 我们会用这个属性来追踪成员关系
        if values is not None:
            for value in values:
                self.add(value)

    def __repr__(self):
        """this is the string representation of a Set object
        if you type it at the Python prompt or pass it to str()"""
        return "Set: " + str(self.dict.keys())

    # 通过成为self.dict中对应值为True的键，来表示成员关系
    def add(self, value):
        self.dict[value] = True

    # 如果它在字典中是一个键，那么在集合中就是一个值
    def contains(self, value):
        return value in self.dict

    def remove(self, value):
        del self.dict[value]
```

可以像下面这样来用上面的函数：

```
s = Set([1,2,3])
s.add(4)
print s.contains(4)        # True
s.remove(3)
print s.contains(3)        # False
```

2.2.7 函数式工具

在传递函数的时候，有时我们可能想部分地应用（或 curry）函数来创建新函数。下面是一个简单的例子，假设我们有一个含两个变量的函数：

```
def exp(base, power):
    return base ** power
```

我们想用它来创建一个单变量的函数 `two_to_the`。它的输入是一个幂次（`power`），输出的是 `exp(2, power)` 的结果。

当然，我们可以用 `def` 来实现，但它有时候使用起来并不太方便：

```
def two_to_the(power):
    return exp(2, power)
```

一个另辟蹊径的方法是使用 `functools.partial`：

```
from functools import partial
two_to_the = partial(exp, 2)        # 现在是一个包含一个变量的函数
print two_to_the(3)                 # 8
```

如果你为后面的参数指定了名字，也能用 `partial` 来填充这些参数：

```
square_of = partial(exp, power=2)
print square_of(3)                    # 9
```

如果你 curry 中间函数的参数，就会变得混乱起来，所以要努力避免这么做。

偶尔我们也会使用函数 map、reduce 和 filter，它们为列表解析提供了函数式替换方案：

```
def double(x):
    return 2 * x

xs = [1, 2, 3, 4]
twice_xs = [double(x) for x in xs]          # [2, 4, 6, 8]
twice_xs = map(double, xs)                  # 和上面一样
list_doubler = partial(map, double)         # double了一个列表的*function*
twice_xs = list_doubler(xs)                 # 同样是[2, 4, 6, 8]
```

如果你提供了多个列表，可以对带有多个参数的函数使用 map：

```
def multiply(x, y): return x * y

products = map(multiply, [1, 2], [4, 5]) # [1 * 4, 2 * 5] = [4, 10]
```

类似地，filter 做了列表解析中 if 的工作：

```
def is_even(x):
    """True if x is even, False if x is odd"""
    return x % 2 == 0

x_evens = [x for x in xs if is_even(x)]         # [2, 4]
x_evens = filter(is_even, xs)                   # 和上面一样
list_evener = partial(filter, is_even)          # filter了一个列表的*function*
x_evens = list_evener(xs)                       # 同样是[2, 4]
```

reduce 结合了列表的头两个元素，它们的结果又结合了列表的第 3 个元素，这个结果之后又结合了第 4 个元素，依次下去，直到得到一个单独的结果：

```
x_product = reduce(multiply, xs)                # = 1 * 2 * 3 * 4 = 24
list_product = partial(reduce, multiply)        # reduce了一个列表的*function*
x_product = list_product(xs)                    # 同样是24
```

2.2.8　枚举

有时候，你可能想在一个列表上迭代，并且同时使用它的元素和元素的索引：

```
# 非Python用法
for i in range(len(documents)):
    document = documents[i]
    do_something(i, document)

# 也非Python用法
i = 0
for document in documents:
    do_something(i, document)
    i += 1
```

Python 惯用的解决方案是使用枚举（enumerate），它会产生 (index, element) 元组：

```
for i, document in enumerate(documents):
    do_something(i, document)
```

类似地，如果你只想要索引，则执行：

```
for i in range(len(documents)): do_something(i)      # 非Python用法
for i, _ in enumerate(documents): do_something(i)    # Python用法
```

我们会频繁用到枚举。

2.2.9 压缩和参数拆分

如果想把两个或多个列表压缩在一起，可以使用 zip 把多个列表转换为一个对应元素的元 组的单个列表中：

```
list1 = ['a', 'b', 'c']
list2 = [1, 2, 3]
zip(list1, list2)          # 是[('a', 1), ('b', 2), ('c', 3)]
```

如果列表的长度各异，zip 会在第一个列表结束时停止。

可以使用一种特殊的方法“解压”一个列表：

```
pairs = [('a', 1), ('b', 2), ('c', 3)]
letters, numbers = zip(*pairs)
```

其中的星号执行参数拆分（argument unpacking）。参数拆分使用 pairs 的元素作为独立的参数传给 zip。这就和调用以下函数的结果是一样的：

```
zip(('a', 1), ('b', 2), ('c', 3))
```

它返回 [('a','b','c'),('1','2','3')]。

你可以在任何函数上使用参数拆分：

```
def add(a, b): return a + b

add(1, 2)      # 返回3
add([1, 2])    # TypeError!
add(*[1, 2])   # 返回3
```

参数拆分并不是特别有用，但当我们用到它的时候，会觉得这是个不错的技巧。

2.2.10 args和kwargs

假如我们想创建一个更高阶的函数，把某个函数 f 作为输入，并返回一个对任意输入都返回 f 值两倍的新函数：

```
def doubler(f):
    def g(x):
        return 2 * f(x)
    return g
```

这个函数在有些情况下可以实现：

```
def f1(x):
    return x + 1

g = doubler(f1)
print g(3)              # 8 (== ( 3 + 1) * 2)
print g(-1)             # 0 (== (-1 + 1) * 2)
```

但对于有多个参数的函数来说，就不适用：

```
def f2(x, y):
```

```
    return x + y

g = doubler(f2)
print g(1, 2)    # TypeError: g()只能有一个参数（给定了两个）
```

我们所需要的是一种指定一个可以取任意参数的函数的方法，利用参数拆分和一点点魔法就可以做到这一点：

```
def magic(*args, **kwargs):
    print "unnamed args:", args
    print "keyword args:", kwargs

magic(1, 2, key="word", key2="word2")

# 输出
# 未命名args: (1, 2)
# 关键词args: {'key2': 'word2', 'key': 'word'}
```

也就是说，当我们定义了这样一个函数时，args 是一个它的未命名参数的元组，而 kwargs 是一个它的已命名参数的 dict。反过来也适用，你可以使用一个 list（或者 tuple）和 dict 来给函数提供参数：

```
def other_way_magic(x, y, z):
    return x + y + z

x_y_list = [1, 2]
z_dict = { "z" : 3 }
print other_way_magic(*x_y_list, **z_dict)    # 6
```

参照以上例子，你可以充分利用这种技巧，随意发挥。我们将会只用它来创建可以将任意参数作为输入的高阶函数：

```
def doubler_correct(f):
    """works no matter what kind of inputs f expects"""
    def g(*args, **kwargs):
        """whatever arguments g is supplied, pass them through to f"""
        return 2 * f(*args, **kwargs)
    return g

g = doubler_correct(f2)
print g(1, 2) # 6
```

2.2.11　欢迎来到DataSciencester

新员工的入职培训到此结束。哦，当然，不要浪费任何所学的东西。

2.3　延伸学习

- Python 的教程比比皆是。官方的教程（https://docs.python.org/2/tutorial/）是不错的入门选择。

- 官方的 IPython 教程（http://ipython.org/ipython-doc/2/interactive/tutorial.html）不太理想，但其教学视频和报告（http://ipython.org/videos.html）还不错。此外，Wes Mckinney 的 *Python for Data Analysis*（O'Reilly，http://shop.oreilly.com/product/0636920023784.do）有一章非常好地讲解了 IPython。

第 3 章　可视化数据

我相信可视化是实现个人目标的最有力的手段之一！

——哈维 · 麦凯

数据可视化是数据科学家工具箱中的一个重要部分。创建可视化很容易，但创建优秀的可视化却很难。

数据可视化有两种主要用途：

- 探索数据
- 交流数据

本章我们将集中探讨你在探索数据和创建可视化过程中所用到的各项技能，可视化将贯穿本书剩余部分。就像本书大部分章节涉及的主题一样，数据可视化是一个非常丰富的研究领域，值得著书阐述。尽管如此，我们还是尝试着告诉你怎样辨别可视化的好坏。

3.1　matplotlib

有许多工具可以用来可视化数据，我们将使用的是应用最广的 matplotlib 库（尽管这暴露了它的年龄。详见 http://matplotlib.org/）。如果你的兴趣是制作用于网络的精良的交互可视化，它可能不是好的选择，但对于条形图、线图和散点图这些简单的图形来说，它很好用。

特别地，我们会使用 matplotlib.pyplot 模块。在最简单的应用中，pyplot 保持着一种内部状态，你可以在其中一步步地创建可视化。一旦创建工作完成，就可以保存（用 savefig()）或显示（用 show()）你的图形。

例如，制作一个（就像图 3-1 这样）非常简单的图形，就可以采取以下步骤：

```
from matplotlib import pyplot as plt

years = [1950, 1960, 1970, 1980, 1990, 2000, 2010]
```

```
gdp = [300.2, 543.3, 1075.9, 2862.5, 5979.6, 10289.7, 14958.3]

# 创建一幅线图，x轴是年份，y轴是gdp
plt.plot(years, gdp, color='green', marker='o', linestyle='solid')

# 添加一个标题
plt.title("名义GDP")

# 给y轴加标记
plt.ylabel("十亿美元")
plt.show()
```

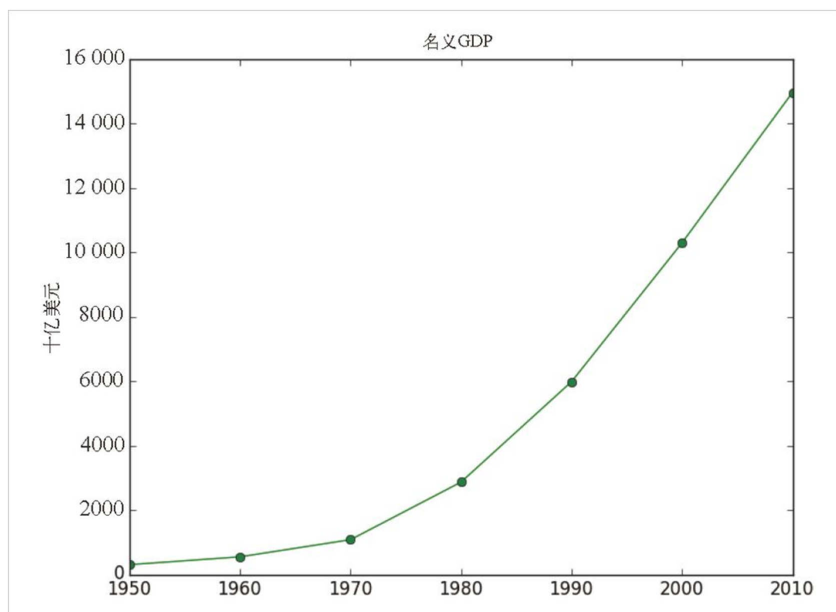


图 **3-1**：一个简单的线图

制作出版级别的精良图片要更加复杂，本章不作深入探讨。对图形进行自定义的方式有多种，如坐标轴标记、线型以及点的形状等。我们并不会面面俱到地讲解这些参数，只会在我们的例子中提到（并关注）其中的一些。

尽管我们并不会用到 `matplotlib` 太多的功能，但它着实能够制作出复杂的图中图、精致的图形格式和交互图形。如果你想做得比本书更深入，请查阅它的文档。

3.2　条形图

如果你想展示某些离散的项目集合中的数量是如何变化的，可以使用条形图。比如，图 3-2 显示了几部电影所获得的奥斯卡金像奖的数目：

```
movies = ["Annie Hall", "Ben-Hur", "Casablanca", "Gandhi", "West Side Stor
num_oscars = [5, 11, 3, 8, 10]

# 条形的默认宽度是0.8，因此我们对左侧坐标加上0.1
# 这样每个条形就被放置在中心了
xs = [i + 0.1 for i, _ in enumerate(movies)]

# 使用左侧x坐标[xs]和高度[num_oscars]画条形图
plt.bar(xs, num_oscars)

plt.ylabel("所获奥斯卡金像奖数量")
plt.title("我最喜爱的电影")

# 使用电影的名字标记x轴，位置在x轴上条形的中心
plt.xticks([i + 0.5 for i, _ in enumerate(movies)], movies)

plt.show()
```

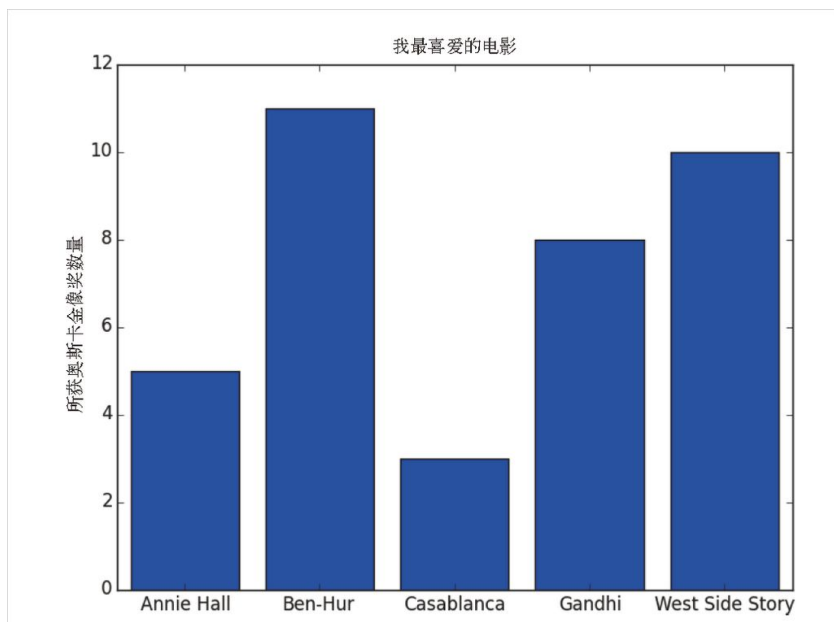


图 **3-2**：一个简单的条形图

条形图也可以用来绘制拥有大量数值取值的变量直方图，以此来探索这些取值是如何分布的，如图 3-3 所示。

```
grades = [83,95,91,87,70,0,85,82,100,67,73,77,0]
decile = lambda grade: grade // 10 * 10
histogram = Counter(decile(grade) for grade in grades)

plt.bar([x - 4 for x in histogram.keys()], # 每个条形向左侧移动4个单位
        histogram.values(),                # 给每个条形设置正确的高度
        8)                                 # 每个条形的宽度设置为8

plt.axis([-5, 105, 0, 5])                  # x轴取值从-5到105
                                           # y轴取值0到5

plt.xticks([10 * i for i in range(11)])    # x轴标记为0，10，...，100
plt.xlabel("十分相")
plt.ylabel("学生数")
plt.title("考试分数分布图")
plt.show()
```

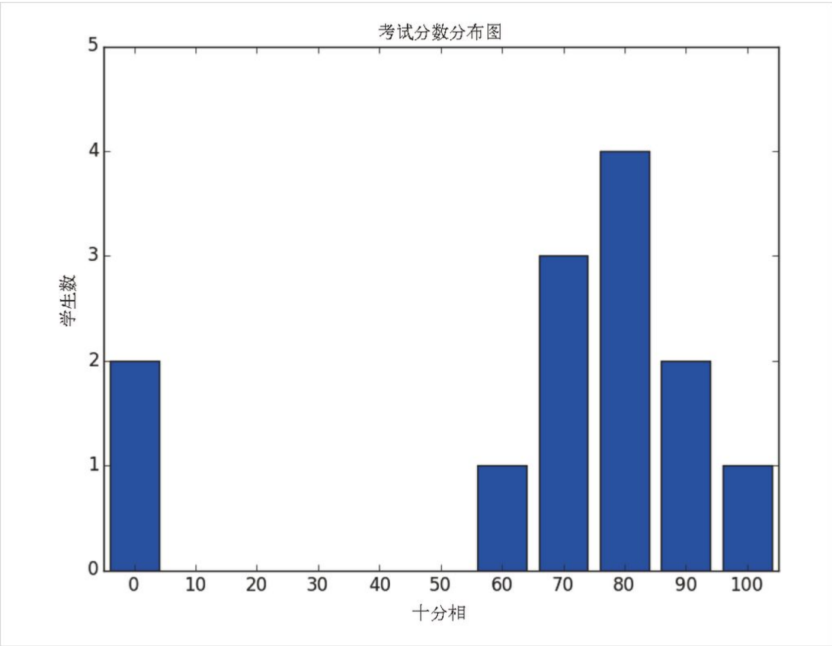


图 **3-3**：为直方图使用条形图

`plt.bar` 的第三个参数指定了条形的宽度，在这里我们选择宽度为 8（这样就在各个条形之间留出了小的间隔，因为 x 轴是以刻度 10 做标记的），而且把每个条形向左移了 4 个宽度。这样一来，（比如说）“80”这个条形的左边在 76，而右边在 84，因此它的中心在 80。

对 `plt.axis` 的调用表明我们希望 x 轴的范围是 -5~105（以使“0”到“100”这些条形可以完全显示），并且 y 轴的范围应限定在 0~5 之间。对 `plt.xticks` 的调用把 x 轴的刻度放在 0、10、20、……、100 这些位置。

在使用 `plt.axis()` 时要谨慎。在创建条形图时，y 轴不从 0 开始是一种特别不好的形式，因为这很容易误导人（见图 3-4）：

```
mentions = [500, 505]
years = [2013, 2014]

plt.bar([2012.6, 2013.6], mentions, 0.8)
plt.xticks(years)
plt.ylabel("听到有人提及‘数据科学’的次数")

# 如果不这么做，matplotlib会把x轴的刻度标记为0和1
# 然后会在角上加上+2.013e3（糟糕的matplotlib操作！）
plt.ticklabel_format(useOffset=False)

# 这会误导y轴只显示500以上的部分
plt.axis([2012.5,2014.5,499,506])
plt.title("快看如此'巨大'的增长！")
plt.show()
```

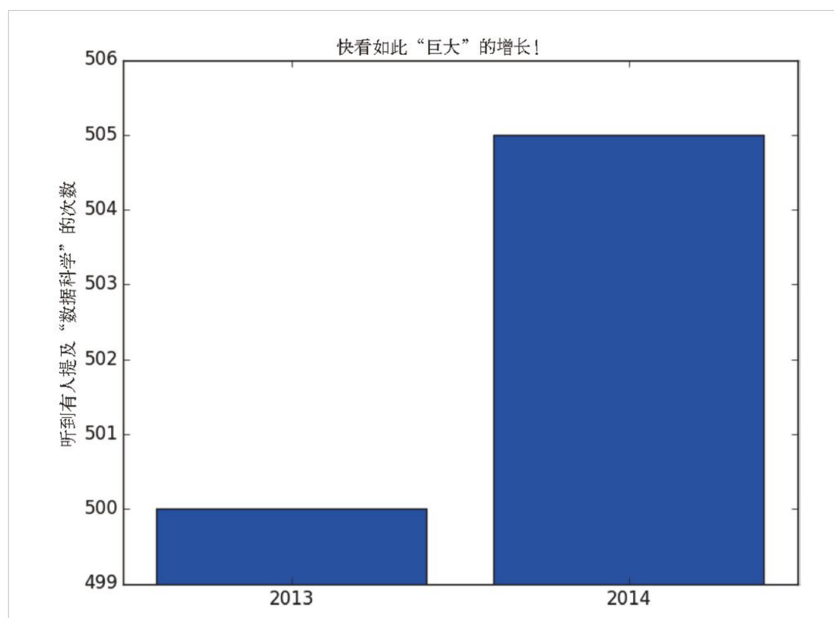


图 **3-4**：一个有误导性 ***y*** 轴的条形图

在图 3-5 中，我们使用了一种更合理的轴，这样它看起来就不那么异常了：

```
plt.axis([2012.5,2014.5,0,550])
plt.title("增长不那么巨大了")
plt.show()
```

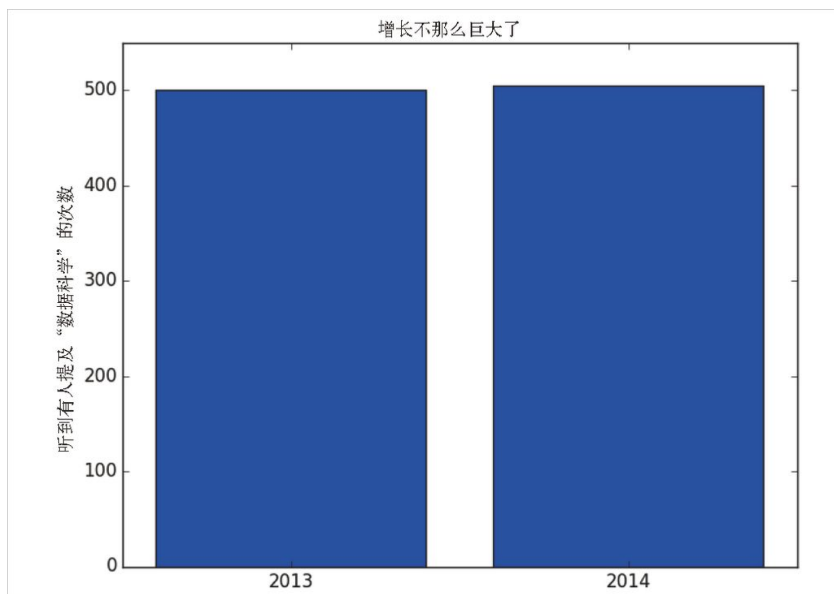


图 3-5：*y* 轴正常的同一个条形图

3.3　线图

前面提过，可以用 `plt.plot()` 来制作线图。这种图形可以用来清晰地显示某种事物的趋势，如图 3-6 所示：

```
variance     = [1, 2, 4, 8, 16, 32, 64, 128, 256]
bias_squared = [256, 128, 64, 32, 16, 8, 4, 2, 1]
total_error  = [x + y for x, y in zip(variance, bias_squared)]
xs = [i for i, _ in enumerate(variance)]

# 可以多次调用plt.plot
# 以便在同一个图上显示多个序列
plt.plot(xs, variance,     'g-',  label='variance')    # 绿色实线
plt.plot(xs, bias_squared, 'r-.', label='bias^2')      # 红色点虚线
plt.plot(xs, total_error,  'b:',  label='total error') # 蓝色点线

# 因为已经对每个序列都指派了标记
# 所以可以自由地布置图例
# loc=9指的是“顶部中央”
plt.legend(loc=9)
plt.xlabel("模型复杂度")
plt.title("偏差-方差权衡图")
plt.show()
```

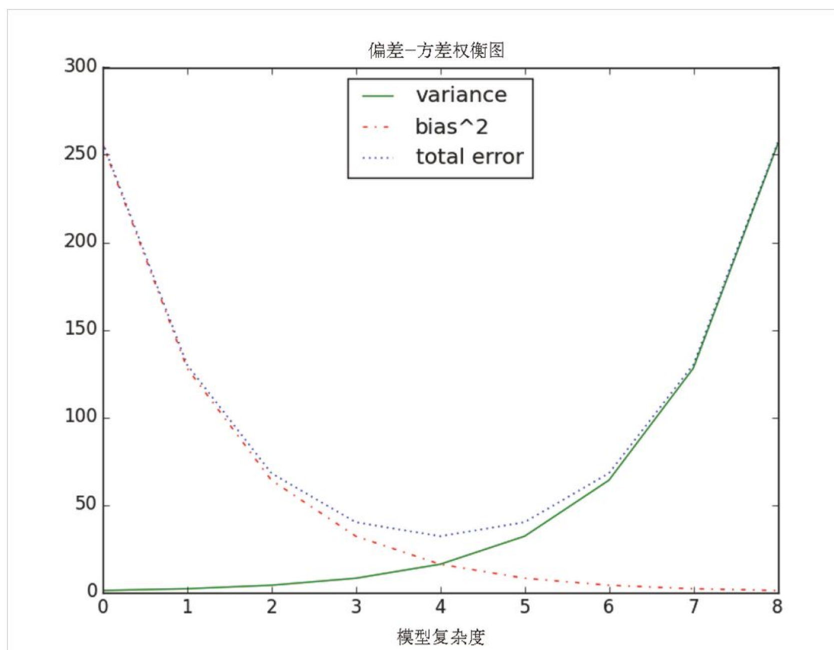


图 3-6：带有图例的几个线图

3.4　散点图

散点图是显示成对数据集的可视化关系的好选择。比如，图 3-7 显示了你的用户们已有的朋友数和他们每天花在网站上的分钟数之间的关系：

```
friends = [ 70,  65,  72,  63,  71,  64,  60,  64,  67]
minutes = [175, 170, 205, 120, 220, 130, 105, 145, 190]
labels =  ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']

plt.scatter(friends, minutes)

# 每个点加标记
for label, friend_count, minute_count in zip(labels, friends, minutes):
    plt.annotate(label,
        xy=(friend_count, minute_count), # 把标记放在对应的点上
        xytext=(5, -5),                  # 但要有轻微偏离
        textcoords='offset points')

plt.title("日分钟数与朋友数")
plt.xlabel("朋友数")
plt.ylabel("花在网站上的日分钟数")
```

```
plt.show()
```

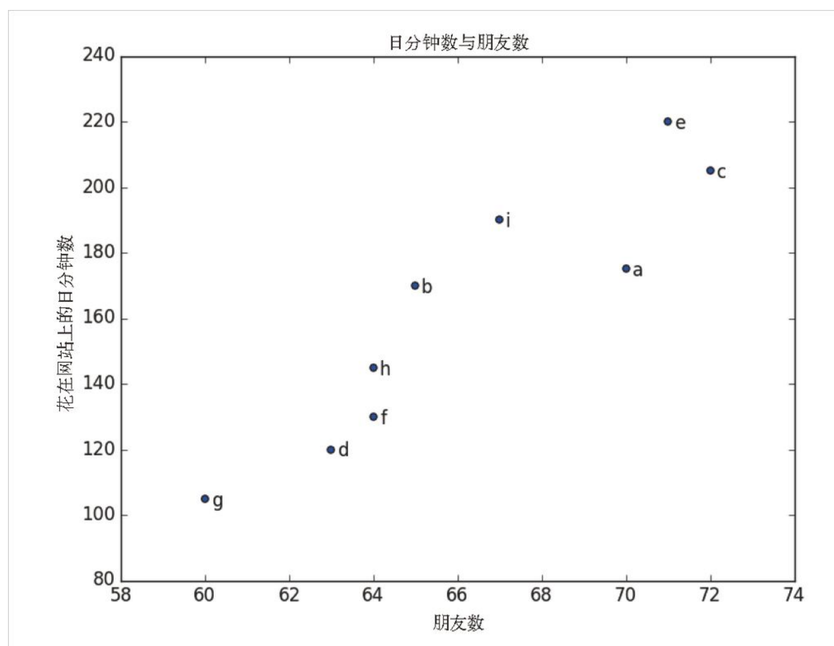


图 **3-7**：朋友数与花在网站上的分钟数之间的关系散点图

当你分散了可比较的变量，如果让 `matplotlib` 选择刻度，可能会得到一个误导性的图，如图 3-8 所示：

```
test_1_grades = [ 99, 90, 85, 97, 80]
test_2_grades = [100, 85, 60, 90, 70]

plt.scatter(test_1_grades, test_2_grades)
plt.title("Axes Aren't Comparable")
plt.xlabel("测验1的分数")
plt.ylabel("测验2的分数")
plt.show()
```

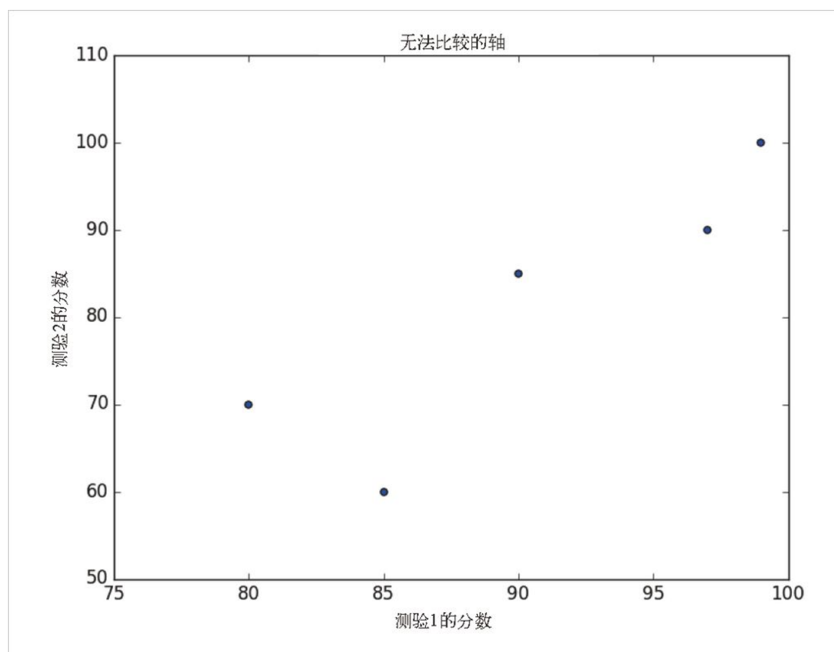


图 **3-8**：带有无法比较的轴的散点图

如果我们引入对 `plt.axis("equal")` 的调用，图形（图 3-9）会更精确地显示大多数变化是发生在测验 2 上的。

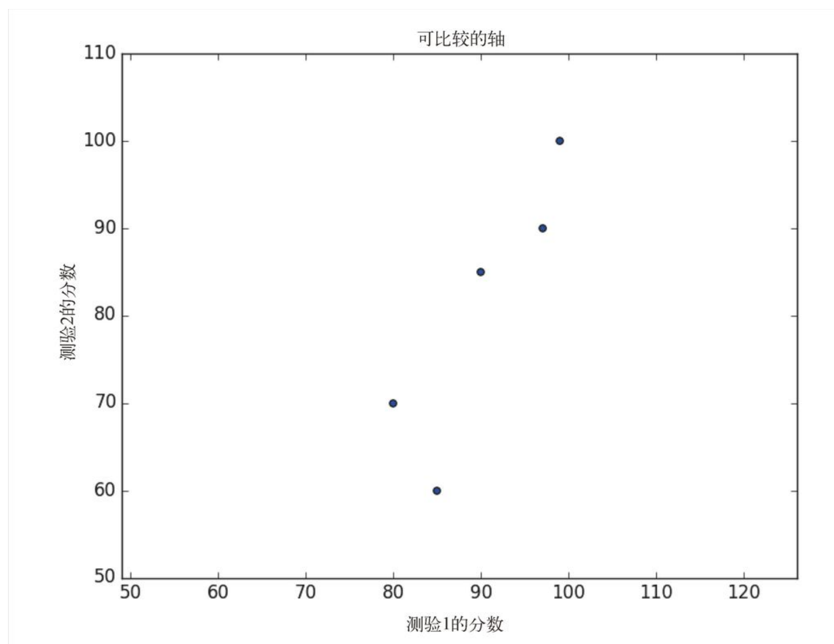


图 **3-9**：带有可比较的轴的同一个散点图

以上这些内容对于开始进行可视化已经足够了，本书后面还会讲解更多关于可视化的知识。

3.5　延伸学习

- seaborn（http://web.stanford.edu/~mwaskom/software/seaborn/）基于 `matplotlib` 构建，让 你可以轻松地制作更漂亮也更复杂的可视化。

- D3.js（http://d3js.org/）是一个 JavaScript 库，可以制作精致的基于网络的交互可视化。尽管它并不存在于 Python 中，但它很时髦，并且应用广泛，值得你花时间来了解它。

- Bokeh（http://bokeh.pydata.org/en/latest/）是一个较新的库，它将 D3 风格的可视化引入了 Python 中。

- ggplot（https://pypi.python.org/pypi/ggplot）是流行的 R 库 `ggplot2` 的 Python 接口。`ggplot2` 被广泛用于生成“出版级别”的图形图像。如果你是一个热情的 `ggplot2` 用户，它可能

是非常有趣的工具，但如果你不熟悉它，可能会觉得使用起来有些困难。

第 4 章　线性代数

百无一用是代数。

——比利 · 康诺利

线性代数是数学的一个分支，研究向量空间。我不指望用一章就能教会你线性代数，但这章涵盖了数据科学的大量概念和技术，因此你至少试着学习一下。本章所学内容在本书后续部分会大量应用。

4.1　向量

抽象地说，向量是指可以加总（以生成新的向量），可以乘以标量（即数字），也可以生成新的向量的对象。

具体来说（对我们而言），向量是有限维空间的点。即使你本无意视你的数据为向量，将数值数据表示为向量也是非常好的处理方式。

比如，如果你有很多人的身高、体重、年龄数据，就可以把数据记为三维向量 (height, weight, age)。如果你教的一个班有四门考试，就可以把学生成绩记为四维向量 (exam1, exam2, exam3, exam4)。

最简单的入门方法是将向量表示为数字的列表。一个包含三个数字的列表对应一个三维空间的向量，反之亦然：

```
height_weight_age = [70,  # 英寸
                     170, # 磅
                     40 ] # 岁

grades = [95,  # 考试1
          80,  # 考试2
          75,  # 考试3
          62 ] # 考试4
```

这种方式的一个问题在于向量算法的应用。由于 Python 中的列表不同于向量（因此无法直接对向量运算），我们需要自己提前构建相应算法工具。现在就开始构建吧！

首先，我们常常需要对两个向量做加法。向量以分量方式（componentwise）做运算。这意味着，如果两个向量 `v` 和 `w` 长度相同，那它们的和就是一个新的向量，其中向量的第一个元素等于 `v[0] + w[0]`，第二个元素等于 `v[1] + w[1]`，以此类推。（如果两个向量长度不同，则不能相加。）

例如，向量 `[1, 2]` 加上向量 `[2, 1]` 等于 `[1 + 2, 2 + 1]` 或 `[3, 3]`，如图 4-1 所示。

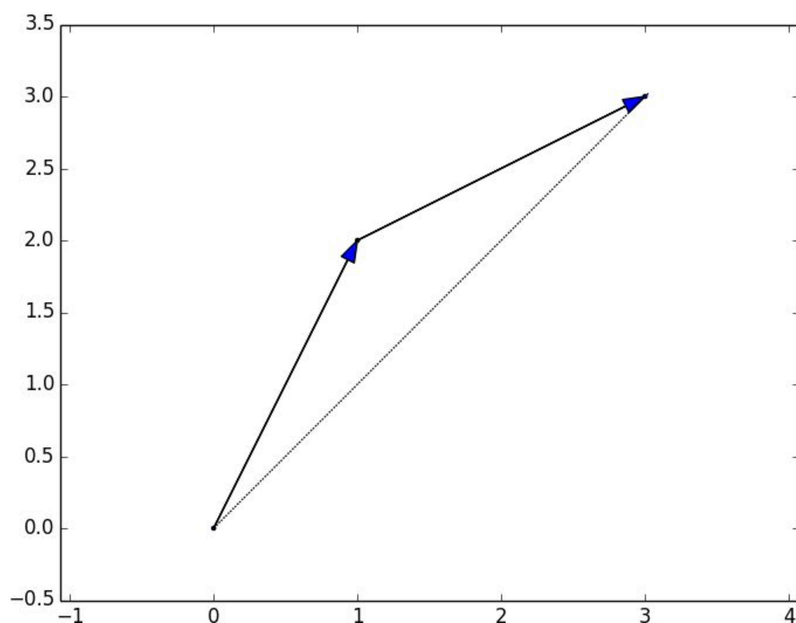


图 **4-1**：两个向量相加

我们可以很容易地实现这个功能：对向量调用 `zip` 函数，同时用列表解析使向量的相应元素相加：

```
def vector_add(v, w):
    """adds corresponding elements"""
    return [v_i + w_i
            for v_i, w_i in zip(v, w)]
```

同样，对两个向量做减法，只需要使向量的相应元素相减：

```
def vector_subtract(v, w):
    """subtracts corresponding elements"""
```

```
    return [v_i - w_i
            for v_i, w_i in zip(v, w)]
```

有时，我们需要对一系列向量做加法。即生成一个新向量，其第一个元素是这一系列向量第一个元素的和，第二个元素是这一系列向量第二个元素的和，以此类推。最简单的方法是每次递加一个向量：

```
def vector_sum(vectors):
    """sums all corresponding elements"""
    result = vectors[0]                               # 从第一个向量开始
    for vector in vectors[1:]:                        # 之后遍历其他向量
        result = vector_add(result, vector)           # 最后计入总和
    return result
```

当你思考这个解决方法时，我们正通过 vector_add 函数，即使用 reduce 的方式来加总这一系列的向量。换句话说，我们用高级的函数更加简洁地实现了这个功能：

```
def vector_sum(vectors):
    return reduce(vector_add, vectors)
```

或者：

```
vector_sum = partial(reduce, vector_add)
```

这最后一种方法很简洁、巧妙，但可能相比之下没有前几种有用处。

当然，我们有时也需要给一个向量乘以一个标量，这时只需将向量的每个元素乘以那个数字：

```
def scalar_multiply(c, v):
    """c is a number, v is a vector"""
    return [c * v_i for v_i in v]
```

我们也可以计算一系列向量（长度相同）的均值：

```
def vector_mean(vectors):
    """compute the vector whose ith element is the mean of the
    ith elements of the input vectors"""
    n = len(vectors)
```

```
    return scalar_multiply(1/n, vector_sum(vectors))
```

一个不常见的功能是点乘（dot product）。两个向量的点乘表示对应元素的分量乘积之和：

```
def dot(v, w):
    """v_1 * w_1 + ... + v_n * w_n"""
    return sum(v_i * w_i
               for v_i, w_i in zip(v, w))
```

点乘衡量了向量 *v* 在向量 *w* 方向延伸的程度。例如，如果 `w=[1, 0]`，则 `dot(v, w)` 就是 `v` 的第一个元素。点乘的另一个解释是将 *v* 在 *w* 上投影所得到的向量的长度（如图 4-2）：

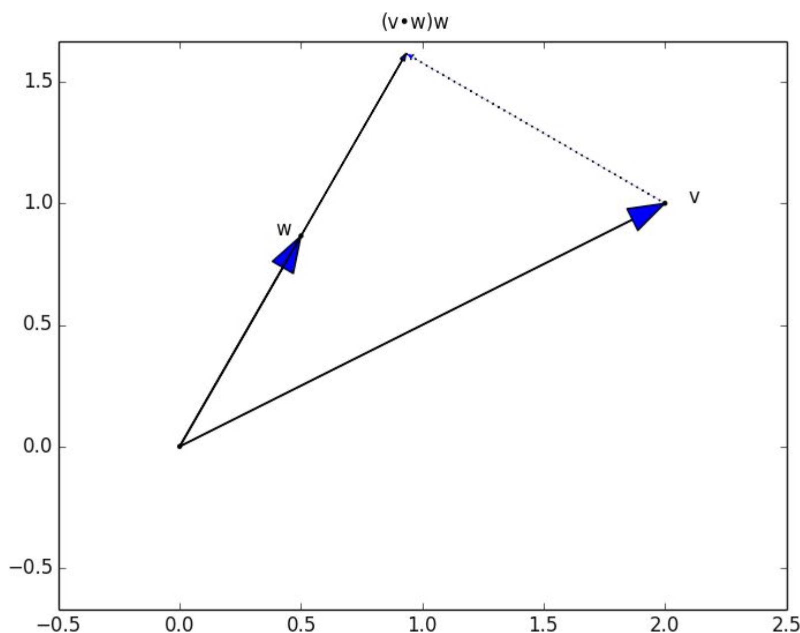


图 **4-2**：点乘即向量投影

通过点乘很容易计算一个向量的平方和：

```
def sum_of_squares(v):
    """v_1 * v_1 + ... + v_n * v_n"""
    return dot(v, v)
```

可以用来计算向量的大小（或长度）：

```
import math

def magnitude(v):
    return math.sqrt(sum_of_squares(v))     # math.sqrt是平方根函数
```

现在，我们得到了为计算两个向量的距离所需要的所有部分，定义如下：

$$\sqrt{(v_1 - w_1)^2 + \cdots + (v_n - w_n)^2}$$

```
def squared_distance(v, w):
    """(v_1 - w_1) ** 2 + ... + (v_n - w_n) ** 2"""
    return sum_of_squares(vector_subtract(v, w))

def distance(v, w):
   return math.sqrt(squared_distance(v, w))
```

写成下式（与上式等价）更清晰：

```
def distance(v, w):
    return magnitude(vector_subtract(v, w))
```

有了这些关于向量的概念和计算，我们就可以开始探讨数据科学了。本书后续部分会大量使用这些概念。



用列表来表示向量很有利于概念阐释，但对性能却影响很糟。

实际编程中，你需要使用 NumPy 库。这个库中有包含各种算法操作的高性能数组类。

4.2　矩阵

矩阵是一个二维的数据集合。我们将矩阵表示为列表的列表，每个内部列表的大小都一样，表示矩阵的一行。如果 `A` 是一个矩阵，那么 `A[i][j]` 就表示第 i 行第 j 列的元素。按照数学表达的惯例，我们通

常用大写字母表示矩阵。例如：

```
A = [[1, 2, 3],    # A有2行3列
     [4, 5, 6]]

B = [[1, 2],       # B有3行2列
     [3, 4],
     [5, 6]]
```


在数学中，矩阵的第一行通常称为“第 1 行”，第一列通常称为“第 1 列”。而我们需要将矩阵的形式和 Python 的列表统一起来：Python 中的列表从 0 开始索引，所以，我们将矩阵的第一行称为“第 0 行”，将第一列称为“第 0 列”。

基于列表的列表这种表达形式，矩阵 A 具有 len(A) 行和 len(A[0]) 列，我们把这称作它的形状（shape）：

```
def shape(A):
    num_rows = len(A)
    num_cols = len(A[0]) if A else 0    # 第一行中元素的个数
    return num_rows, num_cols
```

如果一个矩阵有 n 行 k 列，则可以记为 $n \times k$ 矩阵。我们可以把这个 $n \times k$ 矩阵的每一行都当作一个长度为 k 的向量，把每一列都当作一个长度为 n 的向量：

```
def get_row(A, i):
    return A[i]             # A[i]是第i行

def get_column(A, j):
    return [A_i[j]          # 第A_i行的第j个元素
            for A_i in A]   # 对每个A_i行
```

同样，我们也可以根据形状和用来生成元素的函数来创建矩阵。可以通过一个嵌套的列表解析来实现：

```
def make_matrix(num_rows, num_cols, entry_fn):
    """returns a num_rows x num_cols matrix
    whose (i,j)th entry is entry_fn(i, j)"""
    return [[entry_fn(i, j)             # 根据i创建一个列表
             for j in range(num_cols)]  # [entry_fn(i, 0), ... ]
            for i in range(num_rows)]   # 为每一个i创建一个列表
```

有了这个函数，就可以生成一个 5×5 的单位矩阵（对角线元素是 1，其他元素是 0）：

```
def is_diagonal(i, j):
    """1's on the 'diagonal', 0's everywhere else"""
    return 1 if i == j else 0

identity_matrix = make_matrix(5, 5, is_diagonal)

# [[1, 0, 0, 0, 0],
#  [0, 1, 0, 0, 0],
#  [0, 0, 1, 0, 0],
#  [0, 0, 0, 1, 0],
#  [0, 0, 0, 0, 1]]
```

矩阵的重要性不言而喻，原因有以下几个。

首先，可以通过将每个向量看成是矩阵的一行，来用矩阵表示一个包含多维向量的数据集。例如，如果有 1000 个人的身高、体重和年龄，就可以创建一个 1000×3 的矩阵：

```
data = [[70, 170, 40],
        [65, 120, 26],
        [77, 250, 19],
        # ....
       ]
```

第二，随后我们会看到，可以用一个 $n \times k$ 矩阵表示一个线性函数，这个函数将一个 k 维的向量映射到一个 n 维的向量上。我们使用的很多技术与概念都隐含着这样的函数。

第三，可以用矩阵表示二维关系。在第 1 章中，我们曾经将网络边际表示为数据对 (i, j) 的集合。我们还可以通过建立矩阵 A 来实现这个描述：如果节点 i 和节点 j 有关系，则用矩阵的元素 A[i][j] 为 1 来表示；若节点 i 和节点 j 没有关系，则用 A[i][j] 为 0 来表示。

回想前面讲过的关系：

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (5,
```

可以通过矩阵形式再现：

```
     #      用户 0 1 2 3 4 5 6 7 8 9
     #
friendships = [[0, 1, 1, 0, 0, 0, 0, 0, 0, 0], # 用户 0
               [1, 0, 1, 1, 0, 0, 0, 0, 0, 0], # 用户 1
               [1, 1, 0, 1, 0, 0, 0, 0, 0, 0], # 用户 2
               [0, 1, 1, 0, 1, 0, 0, 0, 0, 0], # 用户 3
               [0, 0, 0, 1, 0, 1, 0, 0, 0, 0], # 用户 4
               [0, 0, 0, 0, 1, 0, 1, 1, 0, 0], # 用户 5
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # 用户 6
               [0, 0, 0, 0, 0, 1, 0, 0, 1, 0], # 用户 7
               [0, 0, 0, 0, 0, 0, 1, 1, 0, 1], # 用户 8
               [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]] # 用户 9
```

如果关系很少，这种表示形式的效率就会很低，因为必须存储很多零。但是，通过矩阵表示可以快速地查找确认某两个节点是否是连接的——通过一个矩阵查找函数即可，不必（有可能会）搜索每条边：

```
friendships[0][2] == 1  # true，0和2是朋友
friendships[0][8] == 1  # false，0和8不是朋友
```

同样，若想找到一个节点的所有连接，只需检查这个节点所在的列（或行）：

```
friends_of_five = [i                                               # 仅需在
                   for i, is_friend in enumerate(friendships[5])  # 一行中
                   if is_friend]                                   # 查找
```

为了加速这个搜寻过程，我们之前给每个节点对象都添加了一个连接列表。但对于太大的、扩展性强的图形来说，成本太高，维护也很难。

本书随后还会探讨这些矩阵。

4.3　延伸学习

- 线性代数在数据科学家中应用很广（通常是隐式使用，且不谙此道的人也经常使用）。阅读这方面的教材是个好主意。网上也有很多免费资源，如：

    - UC Davis 提供的有关线性代数的资源（https://www.math.ucdavis.edu/~linear/）；
    - 圣迈克尔大学提供的有关线性代数的资源（http://joshua.smcvt.edu/linearalgebra/）；
    - 对喜欢挑战的读者来说，*Linear Algebra Done Wrong*（http://www.math.brown.edu/~treil/papers/LADW/LADW.html）是一本进阶读物。
- 如果你使用 NumPy（http://www.numpy.org/）的话，我们此处创建的所有工具都可以免费使用（你赚到了）。

第 5 章　统计学

事实如磐石，统计似蒲草。

——马克 · 吐温

统计学是我们赖以理解数据的数学与技术。这是一个庞大繁荣的领域，要用图书馆的一整个书架（甚至一整间屋子）的书来阐述，绝非一本书的一个章节所能尽述，所以本书的探讨必然不会太深入。我在此抛砖引玉，讲述一些刚好能激发你的兴趣的内容，供你自行深入学习与开拓。

5.1　描述单个数据集

凭借口碑与运气，DataSciencester 已经发展了数十名成员。这时，融资部门的副总来问你要一些关于你的成员有多少朋友的描述，以此来确定他潜在的电梯演说对象。

运用第 1 章中学到的技术可以很容易地生成这个数据。但你现面临的问题是如何描述它。

对任何数据集，最简单的描述方法就是数据本身：

```
num_friends = [100, 49, 41, 40, 25,
               # ......等等许多
              ]
```

对足够小的数据集来说，这甚至可以说是最好的描述方法。但随着数据规模变大，这就显得笨拙又含混了。（想象一个包含一亿个数字的列表。）为此，我们使用统计来提炼和表达数据的相关特征。

首先，我们通过 `Couner` 和 `plt.bar()` 把你的朋友数绘成直方图（图 5-1）：

```
friend_counts = Counter(num_friends)
xs = range(101)                         # 最大值是100
ys = [friend_counts[x] for x in xs]     # height刚好是朋友的个数
plt.bar(xs, ys)
plt.axis([0, 101, 0, 25])
```

```
plt.title("朋友数的直方图")
plt.xlabel("朋友个数")
plt.ylabel("人数")
plt.show()
```

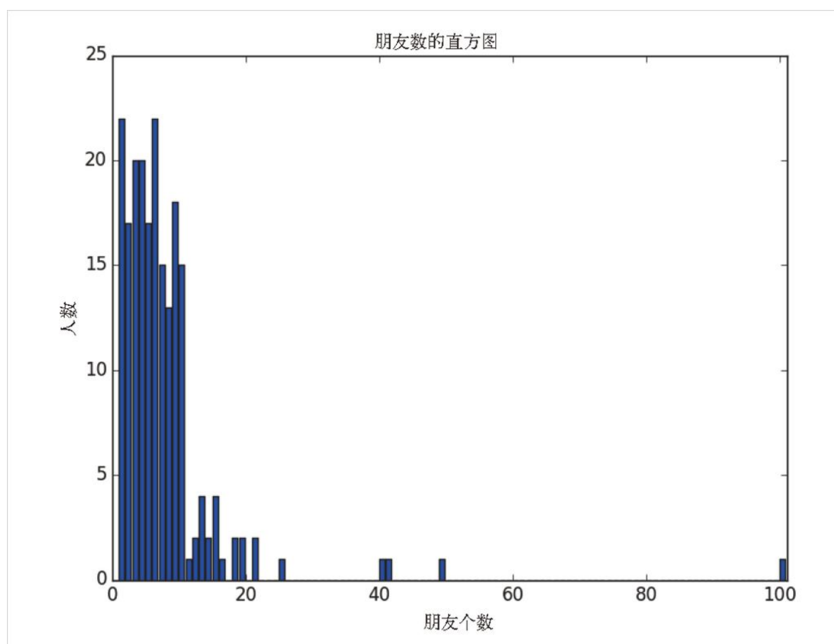


图 **5-1**：朋友数的直方图

不幸的是，这幅图难以用来进行交流，所以你需要再提炼一些统计量。数据点个数大概就是最简单的统计量了：

```
num_points = len(num_friends)               # 204
```

也许你会对数据集的最大值和最小值感兴趣：

```
largest_value = max(num_friends)            # 100
smallest_value = min(num_friends)           # 1
```

如果你想知道特定位置的值，可以这样做：

```
sorted_values = sorted(num_friends)
smallest_value = sorted_values[0]           # 1
```

```
second_smallest_value = sorted_values[1]   # 1
second_largest_value = sorted_values[-2]   # 49
```

当然，这仅仅是开始。

5.1.1 中心倾向

我们常常希望了解数据中心位置的一些概念。一个常用的方法是使用均值（mean 或 average），即用数据和除以数据个数：

```
# 如果没有从__future__导入division，那就是不对的
def mean(x):
    return sum(x) / len(x)

mean(num_friends)   # 7.333333
```

如果你有两个数据点，均值就意味着两点的中间点。随着数据集中点数的增加，均值点会移动，但它始终取决于每个点的取值。

我们常常也会用到中位数（median），它是指数据中间点的值（如果数据点的个数是奇数），或者中间两个点的平均值（如果数据点的个数是偶数）。

例如，如果在排序向量 `x` 上有五个数据点，那么中位数就是 `x[5 // 2]` 或 `x[2]`。如果有六个数据点，则中位数是 `x[2]`（第三个点）与 `x[3]`（第四个点）的平均数。

注意——和均值不同——中位数并不依赖于每一个数据的值。例如，即便数据集中最大的点变得更大（或最小的点变得更小），中间的数据点都不会变，意味着中位数也不会变。

`median` 函数很可能比你想象的更复杂一些，主要是因为数据集中数据个数奇偶性的不同：

```
def median(v):
    """finds the 'middle-most' value of v"""
    n = len(v)
    sorted_v = sorted(v)
    midpoint = n // 2

    if n % 2 == 1:
        # 如果是奇数，返回中间值
        return sorted_v[midpoint]
```

```
    else:
        # 如果是偶数，返回中间两个值的均值
        lo = midpoint - 1
        hi = midpoint
        return (sorted_v[lo] + sorted_v[hi]) / 2
        median(num_friends) # 6.0
```

很明显，均值的计算更简单，并且它会随着数据变化而平稳地变化。如果有 n 个数据点，其中某一个点的值增加了 e，则均值随之增加 e/n。（这使得均值适用于各种微分运算）但是为了计算中位数，得先对数据排序。并且，如果其中一个数据点的值增加了 e，那么中位数有可能也增加 e，有可能增加一个小于 e 的数，也有可能根本不变（这取决于其他的数据）。


事实上，不排序也可以使用一些生僻的技巧有效地算出中位数（https://en.wikipedia.org/wiki/Quickselect）。但这些技巧不但不易理解，而且超出了本书的讲解范围。所以我们先排序再计算。

同时，均值对数据中的异常值非常敏感。如果最具人缘的用户有 200 个朋友（不是 100），均值会上升至 7.82，而中位数不变。如果异常值属于不良数据（或者对我们试图理解的现象不具有代表性），那么均值会误导我们。举一个老生常谈的例子，20 世纪 80 年代，北卡罗来纳大学起薪最高的专业是地理学，因为球星迈克尔 · 乔丹曾就读于此，均值计算就包含了这个“异常值”。

中位数的一个泛化概念是分位数（quantile），它表示少于数据中特定百分比的一个值。（中位数表示少于 50% 的数据的一个值。）

```
def quantile(x, p):
    """returns the pth-percentile value in x"""
    p_index = int(p * len(x))
    return sorted(x)[p_index]

quantile(num_friends, 0.10) # 1
quantile(num_friends, 0.25) # 3
quantile(num_friends, 0.75) # 9
quantile(num_friends, 0.90) # 13
```

还有一个不太常用的概念众数（mode），它是指出现次数最多的一个或多个数：

```
def mode(x):
```

```
    """returns a list, might be more than one mode"""
    counts = Counter(x)
    max_count = max(counts.values())
    return [x_i for x_i, count in counts.iteritems()
            if count == max_count]

mode(num_friends)        # 1 和 6
```

但是，最常用的还是均值。

5.1.2　离散度

离散度是数据的离散程度的一种度量。通常，如果它所统计的值接近零，则表示数据聚集在一起，离散程度很小，如果值很大（无论那意味着什么），则表示数据的离散度很大。例如，一个简单的度量是极差（range），指最大元素与最小元素的差：

```
# "range" 在Python中已经有特定的含义，所以我们换一个不同的名字
def data_range(x):
    return max(x) - min(x)

data_range(num_friends) # 99
```

极差恰好为零，意味着数据集中最大值和最小值相等，这种情形只有在 x 中的元素全部相同时才会发生，意味着数据没有离散。相反，如果极差很大，说明最大元素比最小元素大很多，数据离散度很高。

和中位数一样，极差也不真正依赖于整个数据集。一个只包含 0 和 100 的数据集，和一个包含 0、1 以及很多个 50 的数据集，两者的极差相同。但看起来第一个数据集的离散度“应该”更高。

离散度的另一个更复杂的度量是方差（variance），计算方式如下：

```
def de_mean(x):
    """translate x by subtracting its mean (so the result has mean 0)"""
    x_bar = mean(x)
    return [x_i - x_bar for x_i in x]

def variance(x):
    """assumes x has at least two elements"""
    n = len(x)
    deviations = de_mean(x)
    return sum_of_squares(deviations) / (n - 1)
```

```
variance(num_friends) # 81.54
```

 这个概念看起来似乎是各个数值分别与其均值之差的平方的均值，但我们除以的是 n-1 而不是 n。事实上，如果样本取自更大的总体，x_bar 就是真实均值的估值，意味着 (x_i - x_bar) ** 2 是 x_i 的方差对均值的低估值，所以我们除以 n-1 而不是 n。更多信息请查看维基百科。

现在，无论我们的数据是什么单位（即“朋友”），所有中心倾向的度量都是同一单位。极差的单位也与此相同。但是，方差的单位是原数据单位的平方（即“平方朋友”）。然而，用方差很难给出直观的比较，所以我们更常使用标准差（standard deviation）：

```
def standard_deviation(x):
    return math.sqrt(variance(x))

standard_deviation(num_friends) # 9.03
```

极差和标准差也都有我们之前提到的均值计算常遇到的异常值问题。再看之前的例子，如果我们最具人缘的用户有 200 个朋友，标准差就变为 14.89，增加了 60% ！

一种更加稳健的方案是计算 75% 的分位数和 25% 的分位数之差：

```
def interquartile_range(x):
    return quantile(x, 0.75) - quantile(x, 0.25)

interquartile_range(num_friends) # 6
```

相对来说，这种计算不易受到一小部分异常值的影响。

5.2　相关

DataSciencester 战略发展部的副总持有这样一种想法，即用户在某个网站上花费的时间与其在这个网站上拥有的朋友数相关（他并不是一个无所事事的领导）。现在，他要求你来验证这个想法。

通过分析研究流量日志，你设法做出了一个 daily_minutes 列

表，这个列表描述了每个用户每天在 DataSciencester 花费了多长时间。你还对这个列表排了序，使它的元素和你之前的列表 num_friends 的元素对应了起来，以便进一步研究两个度量之间的关系。

我们先来看一下协方差（covariance），这个概念是方差的一个对应词。方差衡量了单个变量对均值的偏离程度，而协方差衡量了两个变量对均值的串联偏离程度：

```
def covariance(x, y):
    n = len(x)
    return dot(de_mean(x), de_mean(y)) / (n - 1)

covariance(num_friends, daily_minutes) # 22.43
```

回想一下点乘（dot）的概念，它意味着对应的元素对相乘后再求和。如果向量 x 和向量 y 的对应元素同时大于它们自身序列的均值，或者同时小于它们自身序列的均值，那将为求和贡献一个正值。如果其中一个元素大于自身的均值，而另一个小于自身的均值，那将为求和贡献一个负值。因此，如果协方差是一个大的正数，就意味着如果 y 很大，那么 x 也很大，或者如果 y 很小，那么 x 也很小。如果协方差为负而且绝对值很大，就意味着 x 和 y 一个很大，而另一个很小。接近零的协方差意味着以上关系都不存在。

但是，这个数字很难解释，原因如下。

- 它的单位是输入单位的乘积（即朋友 - 分钟 - 每天），难于理解。（“朋友 - 分钟 - 每天”是什么鬼？）

- 如果每个用户的朋友数增加到两倍（但分钟数不变），方差会增加至两倍。但从某种意义上讲，变量的相关度是一样的。换句话讲，很难说“大”的协方差意味着什么。

因此，相关是更常受到重视的概念，它是由协方差除以两个变量的标准差：

```
def correlation(x, y):
    stdev_x = standard_deviation(x)
    stdev_y = standard_deviation(y)
    if stdev_x > 0 and stdev_y > 0:
        return covariance(x, y) / stdev_x / stdev_y
    else:
        return 0    # 如果没有变动，相关系数为零
```

```
correlation(num_friends, daily_minutes) # 0.25
```

相关系数没有单位，它的取值在 -1（完全反相关）和 1（完全相关）之间。相关值 0.25 代表一个相对较弱的正相关。

但是，我们忽略了对数据的检查。看图 5-2。

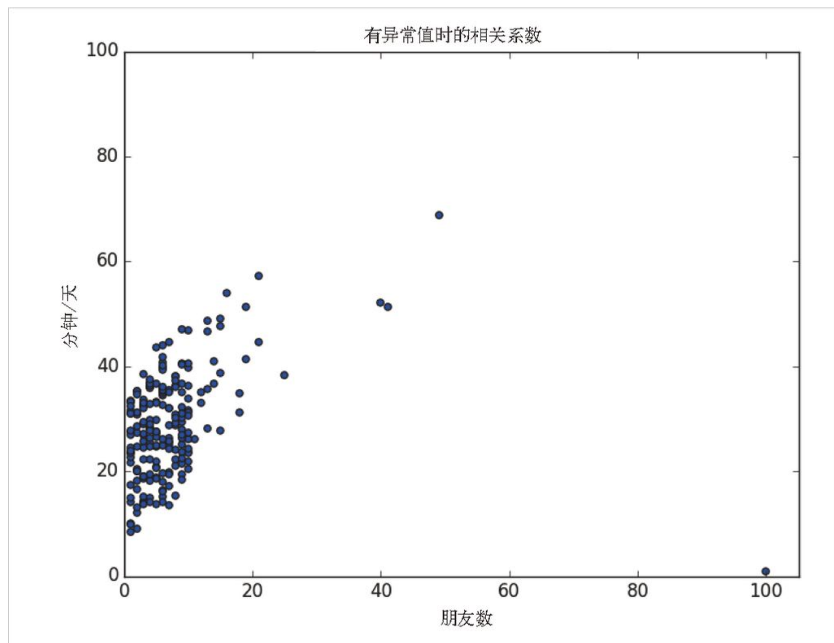


图 5-2：有异常值时的相关系数

图中那个有 100 个朋友的用户（每天只在网上花费 1 分钟）是一个明显的异常值，相关系数的计算对异常值非常敏感。如果我们计算时希望忽略这个人，该怎么做呢？如下所示：

```
outlier = num_friends.index(100) # outlier的索引

num_friends_good = [x
                    for i, x in enumerate(num_friends)
                    if i != outlier]

daily_minutes_good = [x
                      for i, x in enumerate(daily_minutes)
                      if i != outlier]
```

```
correlation(num_friends_good, daily_minutes_good) # 0.57
```

排除了这个异常值，相关性明显增强了（见图 5-3）。

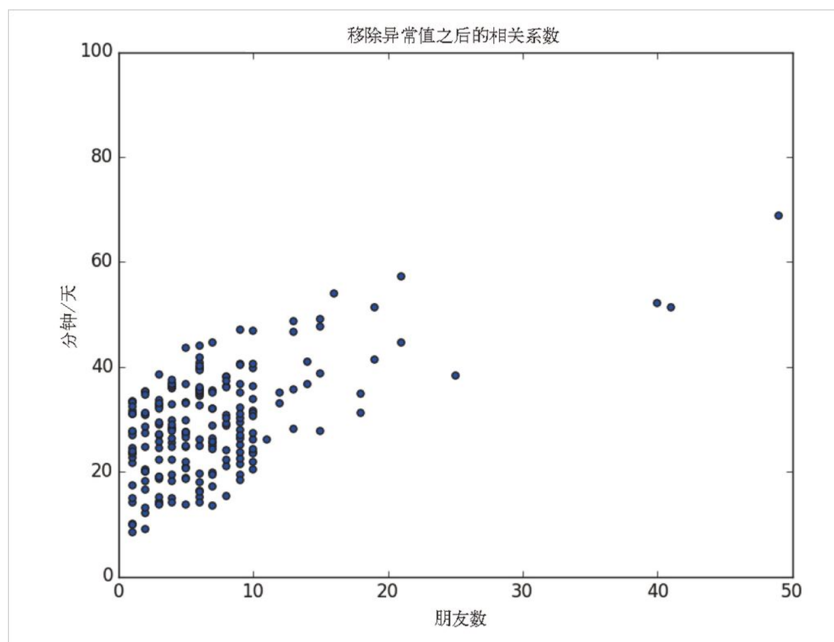


图 **5-3**：移除异常值之后的相关系数

通过进一步调查，你发现这个异常值实际上仅仅是一个内部测试账号，因而没人对移除它有异议。这样你就可以理直气壮地删除它了。

5.3　辛普森悖论

辛普森悖论是指分析数据时可能会发生的意外。具体而言，如果忽略了混杂变量，相关系数会有误导性。

例如，假设你先将所有会员分成东海岸数据科学家和西海岸数据科学家两类，然后决定验证一下哪一边海岸的数据科学家更友好：

	平均朋友数	
西海岸		
东海岸		

很明显，西海岸的数据科学家比东海岸的数据科学家更招人喜欢。你的同事还可以给出许多理由解释这个结果：或许是阳光、咖啡、有机农产品，又或许是旖旎的太平洋风光。

但分析数据时，你却发现了一些奇怪的结论。如果你仅仅比较拥有博士学位的数据科学家，结论表明东海岸数据科学家的平均朋友数更多。如果再仅仅比较没有博士学位的数据科学家，结论仍然是东海岸的数据科学家平均拥有更多的朋友！

[illegible]		[illegible] 平均朋友数		
西海岸 [illegible]				
东海岸 [illegible]				
西海岸 [illegible]				
东海岸 [illegible]				

一旦你考虑了用户的学位，得出的相关系数就会发生变化！将东海岸科学家的数据和西海岸科学家的数据混同起来，会掩盖一件事实，即东海岸数据科学家更偏向博士类型。

这种现象在现实世界中时有发生。关键点在于，相关系数假设在其他条件都相同的前提之下衡量两个变量的关系。而当数据类型变成随机分配，就像置身于精心设计的实验之中时，“其他条件都相同”也许还不是一个糟糕的前提假设。但如果存在另一种类型分配的更深的机制，“其他条件都相同”可能会成为一个糟糕的前提假设。

避免这种窘境的唯一务实的做法是充分了解你的数据，并且尽可能核查所有可能的混杂因素。显然，这不可能万无一失。如果你没有这 200 个数据科学家的受教育程度的数据，你很可能就已经得出了西海岸的数据科学家天生更有社交能力的结论。

5.4　相关系数其他注意事项

相关系数为零表示两个变量之间不存在线性关系。但它们之间还可能会存在其他形式的关系。例如，如果：

```
x = [-2, -1, 0, 1, 2]
y = [ 2,  1, 0, 1, 2]
```

那么，x 和 y 的相关系数为 0。但容易看出，x 和 y 之间显然具有

某种关系——y 中的每个元素等于 x 中相应元素的绝对值。然而，有一种关系它们却无法给出，即 x_i 和 mean(x) 之间的关系与 y_i 和 mean(y) 之间的关系并没有太大关联。这是一种相关系数试图捕捉的关系。

此外，相关系数无法告诉你关系有多强。例如：

```
x = [-2, 1, 0, 1, 2]
y = [99.98, 99.99, 100, 100.01, 100.02]
```

以上这两个变量完全相关，但（这取决于你想度量什么）很有可能这种关系并没有实际意义。

5.5　相关和因果

你很可能听说过这样一句话："相关不是因果。"这样的说辞大致出自一位遇到了一堆威胁着他不可动摇的世界观的数据的人之口。然而，这是个重要的论断——如果 x 和 y 强相关，那么意味着可能 x 引起了 y，或 y 引起了 x，或者两者相互引起了对方，或者存在第三方因素同时引起了 x 和 y，或者什么都不是。

回想一下 num_friends 和 daily_minutes 之间的关系。如果 DataSciencester 用户在网站上拥有更多的朋友，可能会引起一个结果，即这些用户可能就会愿意在网上花费更多的时间。也可能是这种情形：如果每个朋友每天发布一定数量的内容，那么用户的朋友越多，就需要越多的时间来浏览朋友们的更新。

但是，也有这样一种可能。你泡在 DataSciencester 论坛上的时间越长，你就越有可能碰上和结识志同道合的朋友。这也意味着，在网站上花费时间越多，就会拥有更多朋友。

第三种可能是，越是那些热衷于数据科学的用户，就越喜欢在网上花更多时间（因为他们发现这更有趣），并且更乐于结交数据科学家朋友（因为他们对其他人不感冒）。

进行随机试验是证实因果关系的可靠性的一个好方法。你可以先将一组具有类似的统计数据的用户随机分为两组，再对其中一组施加稍微不同的影响因素，然后你会发现，不同的因素会导致不同的结果。

比如，如果你不介意因拿用户做试验而受谴责（http://

www.nytimes.com/2014/06/30/technology/facebook-tinkers-with-users-emotions-in-news-feed-experiment-stirring-outcry.html?_r=0），可以随机从用户中抽取一个小样本，只给他们看他们一小部分朋友的动态更新。如果这个小样本中的用户在网上花费的时间相应地变少，那么你就可以肯定“拥有更多朋友会引起上网时间变长”这一结论了。

5.6　延伸学习

- SciPy（http://docs.scipy.org/doc/scipy/reference/stats.html）、pandas（http://pandas.pydata.org/）和StatsModels（http://statsmodels.sourceforge.net/）等软件都含有丰富的统计函数。

- 统计学非常重要。（或者说，各种统计资料非常重要？）如果你想成为一名优秀的数据科学家，最好先熟读一本统计学教科书。网上有很多免费资源，其中我比较喜欢的有：

    - *OpenIntro Statistics*（https://www.openintro.org/stat/textbook.php）

    - *OpenStax Introductory Statistics*（http://openstaxcollege.org/textbooks/introductory-statistics）

第 6 章　概率

概率法则大致上是真实的，但某些时候会很荒谬。

——爱德华 · 吉本

如果对概率论和相关的数学原理理解得不够多，从事数据科学工作就极其困难。类似于本书第 5 章讲解统计学的方式，我们在本章粗略地讲解一下概率论，基本上不对细节做深入探讨。

为了达到我们的目的，你得把概率论视为对从事件空间中抽取的事件的不确定性进行量化的一种方式。我们暂且不探究术语的技术内涵，而是用掷骰子的例子来理解它们。空间是指所有可能的结果的集合。这些结果的任何一部分就是一个事件，比如，“骰子掷出的点数为 1”“骰子掷出的点数是偶数”等都是事件。

我们用 $P(E)$ 来标记“事件 E 的概率”。

我们接下来用概率理论构建模型，并用概率理论计算模型。概率理论无处不在。

只要你感兴趣，可以深入挖掘概率理论蕴含的哲学原理（边喝啤酒边研究就更好了）。但本书中我们不深入研究。

6.1　不独立和独立

泛泛地讲，如果 E 发生意味着 F 发生（或者 F 发生意味着 E 发生），我们就称事件 E 与事 件 F 为不相互独立（dependent）。反之，E 与 F 就相互独立（independent）。

举个例子，如果两次掷起一枚均匀的硬币，那么我们无法根据第一次掷硬币的结果是否是正面朝上来判断第二次的结果是否正面朝上。第一次掷硬币的结果和第二次掷硬币的结果，这两个事件就是独立的。相反，如果第一次掷硬币的结果是正面朝上，那么我们能很明显能判断出两次掷硬币是否都是反面朝上。（如果第一次掷硬币的结果是正面朝上，那么很明显两次掷硬币的结果不可能都是反面朝上。）因此，第一次结果正面朝上和两次结果不可能都是反面朝上，这两个事件就是不独立的。

从数学角度讲，事件 E 和事件 F 独立意味着两个事件同时发生的概率等于它们分别发生的 概率的乘积：

$$P(E,\ F) = P(E)P(F)$$

在上例中，“第一次掷硬币正面朝上”的概率为 1/2，“两次掷硬币都是背面朝上”的概率为 1/4，但“第一次掷硬币正面朝上并且两次掷硬币都是背面朝上”的概率为 0。

6.2 条件概率

如果事件 E 与事件 F 独立，那么定义式如下：

$$P(E,\ F) = P(E)P(F)$$

如果两者不一定独立（并且 F 的概率不为零），那么 E 关于 F 的条件概率式如下：

$$P(E|F) = P(E,\ F)/P(F)$$

条件概率可以理解为，已知 F 发生，E 会发生的概率。

更常用的公式是上式的变形：

$$P(E,\ F) = P(E|F)P(F)$$

如果 E 和 F 独立，则上式应该表示为：

$$P(E|F) = P(E)$$

这个数学公式意味着，F 是否发生并不会影响 E 是否发生的概率。

举一个常见的关于一个有两个孩子（性别未知）的家庭的有趣例子。

如果我们假设：

(1) 每个孩子是男孩和是女孩的概率相同

(2) 第二个孩子的性别概率与第一个孩子的性别概率独立

那么，事件“没有女孩”的概率是 1/4，事件“一个男孩，一个女孩”的概率为 1/2，事件“两个女孩”的概率为 1/4。

现在，我们的问题是，事件 B“两个孩子都是女孩”关于事件 G“大孩子是女孩”的条件概率是多少？用条件概率的定义式进行计算如下：

$$P(B|G) = P(B,\ G)/P(G) = P(B)/P(G) = 1/2$$

事件 B 与 G 的交集（“两个孩子都是女孩并且大孩子是女孩”）刚好是事件 B 本身。（一旦你知道两个孩子都是女孩，那大孩子必然是女孩。）

这个结果大致上符合你的直觉。

我们接着再问，事件“两个孩子都是女孩”关于事件“至少一个孩子是女孩”（L）的条件概率是多少？出乎意料的是，结果异于前问。

与前问相同的是，事件 B 和事件 L 的交集（“两个孩子都是女孩，并且至少一个孩子是女孩”）刚好是事件 B。这意味着：

$$P(B|L) = P(B,\ L)/P(L) = P(B)/P(L) = 1/3$$

为什么会有这样的结果？如果你已知至少一个孩子是女孩，那么这个家庭有一个男孩和一个女孩的概率是有两个女孩的两倍。

我们可以通过“生成”许多家庭来验证这个结论：

```
def random_kid():
    return random.choice(["boy", "girl"])

both_girls = 0
older_girl = 0
either_girl = 0

random.seed(0)
for _ in range(10000):
    younger = random_kid()
    older = random_kid()
    if older == "girl":
        older_girl += 1
    if older == "girl" and younger == "girl":
        both_girls += 1
    if older == "girl" or younger == "girl":
        either_girl += 1

print "P(both | older):", both_girls / older_girl # 0.514 ~ 1/2
print "P(both | either): ", both_girls / either_girl # 0.342 ~ 1/3
```

6.3　贝叶斯定理

贝叶斯定理是数据科学家的最佳朋友之一，它是条件概率的某种逆运算。假设我们需要计算事件 E 基于已发生的事件 F 的条件概率，但我们已知的条件仅仅是事件 F 基于已发生的事件 E 的条件概率。两次利用条件概率的定义，可以得到下式：

$$P(E|F) = P(E,\ F)/P(F) = P(F|E)P(E)/P(F)$$

事件 F 可以分割为两个互不重合的事件"F 和 E 同时发生"与"F 发生 E 不发生"。我们用符号 $\neg E$ 指代"非 E"（即"E 没有发生"），有下式：

$$P(F) = P(F,\ E) + P(F, \neg E)$$

因此：

$$P(E|F) = P(F|E)P(E)/[P(F|E)P(E) + P(F|\neg E)P(\neg E)$$

上式为贝叶斯定理常用的表达方式。

贝叶斯定理常常用来证明为什么数据科学家比医生更聪明。假设有这样一种病，10 000 个人中会有一个得这个病。还假设有种针对该病的测试，具有 99% 的可能性能给出正确判断（如果患病，测试显示"有病"，如果健康，则显示"无病"）。

阳性的测试结果意味着什么呢？我们用 T 表示"测试结果阳性"，用 D 表示"你患有该病"。那么，根据贝叶斯定理，如果测试结果为阳性，那么你患有该病的概率是：

$$P(D|T) = P(T|D)P(D)/[P(T|D)P(D) + P(T|\neg D)P(\neg D)$$

我们知道，$P(T|D)$，即一个人测试结果为阳性并且本人实际患病的概率为 0.99。$P(D)$，即一个人实际患病的概率是 1/10 000＝0.0001。$P(T|\neg D)$，即一个不患病的人检测结果呈阳性的概率是 0.01 。$P(\neg D)$，即一个人实际上不患该病的概率为 0.9999 。如果将以上数据代入贝叶斯定理，可得：

$$P(D|T) = 0.98$$

结果表示，测试结果为阳性的人实际患病的概率不到 1%。

 移除异常值之后的相关系数我们实际上假设了人们接受测试的概率或多或少都是随机的。如果只有表现出特定症状的人才会去接受测试，我们应该将表达式重新表达为基于事件“测试结果正常，并且表现出症状”的条件概率，这样计算出的结果会增高很多。

对于数据科学家来说，这是小菜一碟，但大部分医生会猜测 $P(D|T)$ 的值接近 2。

一个更直观的计算方式是，首先假设总体包括 1 百万个人。你预期其中 100 个人患有该病，而这 100 个人中会有 99 个测试结果显示阳性。另一方面，你认为 999 900 个人不患有该病，其中 9999 个人测试结果呈阳性。这意味着在（99+9999）个测试结果呈阳性的人中，你认为仅有 99 个人实际上患有该病。

6.4　随机变量

随机变量指这样一种变量，其可能的取值有一种联合概率分布。定义一个简单的随机变量：掷起一枚硬币，如果正面朝上，随机变量等于 1，如果背面朝上，随机变量等于 0。可以再定义更复杂些的随机变量，如掷起一枚硬币 10 次，查看正面朝上的次数，或者从 `range(10)` 取出的一个值，每个数值被取到的可能性都相等。

联合分布对变量实现每种可能值都赋予了概率。通过掷硬币得到的随机变量等于 0 的概率为 0.5，等于 1 的概率也为 0.5。从 `range(10)` 中生成随机变量的分布为从 0 到 9 之间的每个数值赋予 0.1 的概率。

我们有时会讨论一个随机变量的期望值，表示这个随机变量可能值的概率加权值。掷硬币随机变量的期望值为 1/2(=0 * 1/2 + 1 * 1/2)，而 `range(10)` 随机变量的期望值为 4.5。

随机变量也可以基于某些条件事件产生，就像其他事件一样。回忆 6.2 节“条件概率”中提到的双生子例子：如果 X 是表示女孩个数的随机变量，那么 X 等于 0 的概率为 1/4 ，等于 1 的概率为 1/2 ，等于 2 的概率为 1/4。

在已知至少一个孩子是女孩后，接着再定义一个新的随机变量 Y，表示基于这个条件的女孩的个数。Y 等于 1 的概率为 2/3，等于 2 的概率为 1/3 。还定义另一个新的随机变量 Z，表示基于大孩子是女孩的

条件之上的女孩的个数，Z 等于 1 的概率为 1/2 ，等于 2 的概率为 1/2。

大部分情况下，我们会隐式使用随机变量的概念，即在使用时，并没有对此加以特别关注。如果仔细思考，可以看出其中的端倪。

6.5　连续分布

掷硬币对应的是离散分布（discrete distribution）——对离散的结果赋予正概率。我们常常希望对连续结果的分布进行建模。（对于我们的研究目的来说，这些结果最好都是实数，但实际中并不总是这样的）例如，均匀分布（uniform distribution）函数对 0 到 1 之间的所有值都赋予相同的权重（weight）。

因为 0 和 1 之间有无数个数字，因而对每个点而言，赋予的权重几乎是零。因此，我们用带概率密度函数（probability density function，pdf）的连续分布来表示概率，一个变量位于某个区间的概率等于概率密度函数在这个区间上的积分。



如果积分运算不直观，有一种更简单的理解方式：一个分布的密度函数为 f，如果 h 很小，则变量的值落在 x 与 $x+h$ 之间的概率接近 $h* f(x)$。

均匀分布的密度函数如下：

```
def uniform_pdf(x):
    return 1 if x >= 0 and x < 1 else 0
```

如你预期的，一个服从均匀分布的随机变量落在 0.2 和 0.3 之间的概率为 1/10。在 Python 中，`random.random()` 是按均匀分布生成的伪随机数的函数。

我们还常常对累积分布函数（cumulative distribution function，cdf）感兴趣，这个函数给出了一个随机变量小于等于某一特定值的概率。生成均匀分布的累积分布函数不难（见图 6-1）：

```
def uniform_cdf(x):
    "returns the probability that a uniform random variable is <= x"
    if x < 0:   return 0    # 均匀分布的随机变量不会小于0
    elif x < 1: return x    # e.g. P(X <= 0.4) = 0.4
    else:       return 1    # 均匀分布的随机变量总是小于1
```

```
```

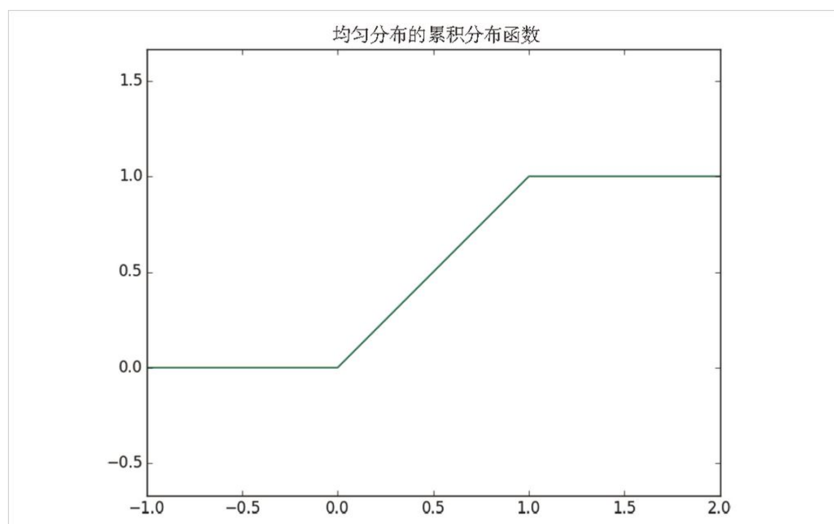


图 6-1：均匀分布的累积分布函数

6.6 正态分布

正态分布是分布之王！它是典型的钟型曲线形态分布函数，可以完全由两个参数决定：均值 μ（mu）和标准差 σ（sigma）。均值描述钟型曲线的中心，标准差描述曲线有多“宽”。

正态分布的分布函数如下：

$$f(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

我们可以这样实现：

```
def normal_pdf(x, mu=0, sigma=1):
    sqrt_two_pi = math.sqrt(2 * math.pi)
    return (math.exp(-(x-mu) ** 2 / 2 / sigma ** 2) / (sqrt_two_pi * sigma
```

在图 6-2 中我们绘出了这些概率密度函数，来看看它们的形状如何：

```
xs = [x / 10.0 for x in range(-50, 50)]
```

```
plt.plot(xs,[normal_pdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_pdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_pdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.
plt.plot(xs,[normal_pdf(x,mu=-1)    for x in xs],'-.',label='mu=-1,sigma=1'
plt.legend()
plt.title("多个正态分布的概率密度函数")
plt.show()
```

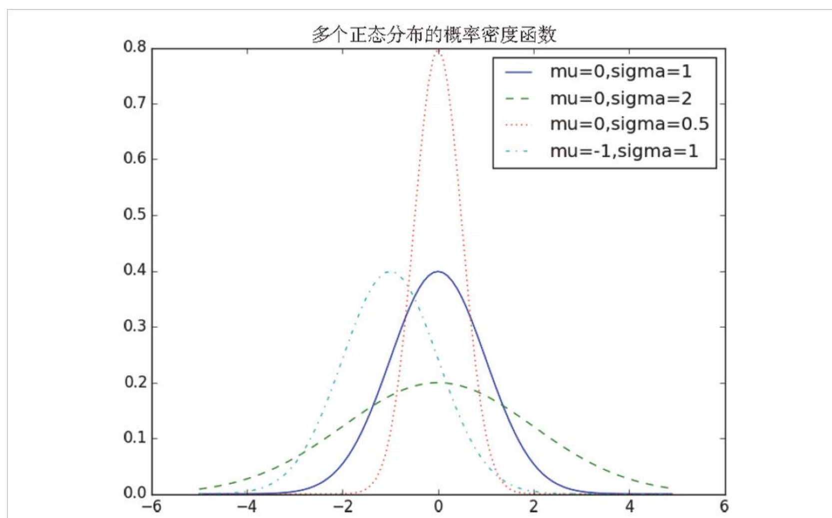


图 **6-2**：多个正态分布的概率密度函数

如果 $\mu=0$ 并且 $\sigma=1$，这个分布称为标准正态分布。如果 Z 是服从标准正态分布的随机变量，则有如下转换式：

$$X = \sigma Z + \mu$$

其中 X 也是正态分布，但均值是 μ，标准差是 σ。相反，如果 X 是均值为 μ 标准差为 σ 的正态分布，那么：

$$Z = (X - \mu)/\sigma$$

是标准正态分布的随机变量。

标准正态分布的累积分布函数无法用“基本”的解析形式表示，但在 Python 中可以用函数 `math.erf` 描述：

```
def normal_cdf(x, mu=0,sigma=1):
    return (1 + math.erf((x - mu) / math.sqrt(2) / sigma)) / 2
```

我们再绘出一系列概率累积分布函数（如图 6-3）：

```
xs = [x / 10.0 for x in range(-50, 50)]
plt.plot(xs,[normal_cdf(x,sigma=1) for x in xs],'-',label='mu=0,sigma=1')
plt.plot(xs,[normal_cdf(x,sigma=2) for x in xs],'--',label='mu=0,sigma=2')
plt.plot(xs,[normal_cdf(x,sigma=0.5) for x in xs],':',label='mu=0,sigma=0.
plt.plot(xs,[normal_cdf(x,mu=-1) for x in xs],'-.',label='mu=-1,sigma=1')
plt.legend(loc=4) # 底部右边
plt.title("多个正态分布的累积分布函数")
plt.show()
```

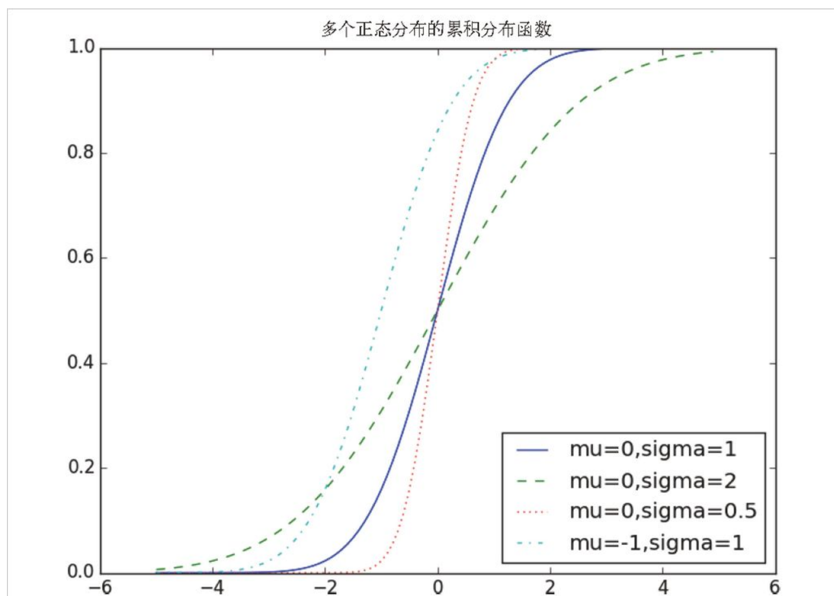


图 **6-3**：多个正态分布的累积分布函数

我们有时会需要对 `normal_cdf` 取逆，从而可以求出特定的概率的相应值。不存在计算逆函数的简便方法，但由于 `normal_cdf` 连续且严格递增，因而我们可以使用二分查找（https://en.wikipedia.org/wiki/Binary_search_algorithm）的方法：

```
def inverse_normal_cdf(p, mu=0, sigma=1, tolerance=0.00001):
    """find approximate inverse using binary search"""
    # 如果非标准型，先调整单位使之服从标准型
    if mu != 0 or sigma != 1:
        return mu + sigma * inverse_normal_cdf(p, tolerance=tolerance)
```

```
    low_z, low_p = -10.0, 0            # normal_cdf(-10)是(非常接近)0
    hi_z,  hi_p  = 10.0, 1             # normal_cdf(10)是(非常接近)1
    while hi_z - low_z > tolerance:
        mid_z = (low_z + hi_z) / 2     # 考虑中点
        mid_p = normal_cdf(mid_z)      # 和cdf在那里的值
        if mid_p < p:
            # midpoint仍然太低，搜索比它大的值
            low_z, low_p = mid_z, mid_p
        elif mid_p > p:
            # midpoint仍然太高，搜索比它小的值
            hi_z, hi_p = mid_z, mid_p
        else:
            break

    return mid_z
```

这个函数反复分割区间，直到分割到一个足够接近于期望概率的精细的 Z 值。

6.7 中心极限定理

正态分布的运用如此广泛，很大程度上归功于中心极限定理（central limit theorem）。这个定理说，一个定义为大量独立同分布的随机变量的均值的随机变量本身就是接近于正态分布的。

特别地，如果 $x_1, \cdots, x_n$ 都是均值为 μ、标准差为 σ 的随机变量，且 n 很大，那么：

$$\frac{1}{n}(x_1 + \cdots + x_n)$$

近似正态分布，且均值为 μ，标准差为 $\sigma/\sqrt{n}$。等价于（其实更常用）：

$$\frac{(x_1 + \cdots + x_n) - \mu n}{\sigma\sqrt{n}}$$

上式近似正态分布，均值为 0 ，标准差为 1。

举一个易于理解的验证例子——带有 n 和 p 两个参数的二项式随机变量。一个二项式随机变量 Binonimal(n,p) 是 n 个独立伯努利随机变量 Bernoulli(p) 之和，每个伯努利随机变量等于 1 的概率是 p，等于

0 的概率是 1-*p*：

```
def bernoulli_trial(p):
    return 1 if random.random() < p else 0

def binomial(n, p):
    return sum(bernoulli_trial(p) for _ in range(n))
```

每个伯努利随机变量 Bernoulli(p) 的均值为 *p*，标准差为 $\sqrt{p(1-p)}$。根据中心极限定理，当 *n* 变得很大，一个二项式随机变量 Binonimal(n,p) 近似于一个正态分布的随机变量，其中均值为 $\mu=np$，标准差为 $\sigma=\sqrt{np(1-p)}$。如果把两个分布都在图上绘出来，很容易看出相似性：

```
def make_hist(p, n, num_points):

    data = [binomial(n, p) for _ in range(num_points)]

    # 用条形图绘出实际的二项式样本
    histogram = Counter(data)
    plt.bar([x - 0.4 for x in histogram.keys()],
            [v / num_points for v in histogram.values()],
            0.8,
            color='0.75')

    mu = p * n
    sigma = math.sqrt(n * p * (1 - p))

    # 用线形图绘出正态近似
    xs = range(min(data), max(data) + 1)
    ys = [normal_cdf(i + 0.5, mu, sigma) - normal_cdf(i - 0.5, mu, sigma)
          for i in xs]
    plt.plot(xs,ys)
    plt.title("二项分布与正态近似")
    plt.show()
```

比如，若调用函数 `make_hist(0.75, 100, 10000)`，可以得到图 6-4 中的图。

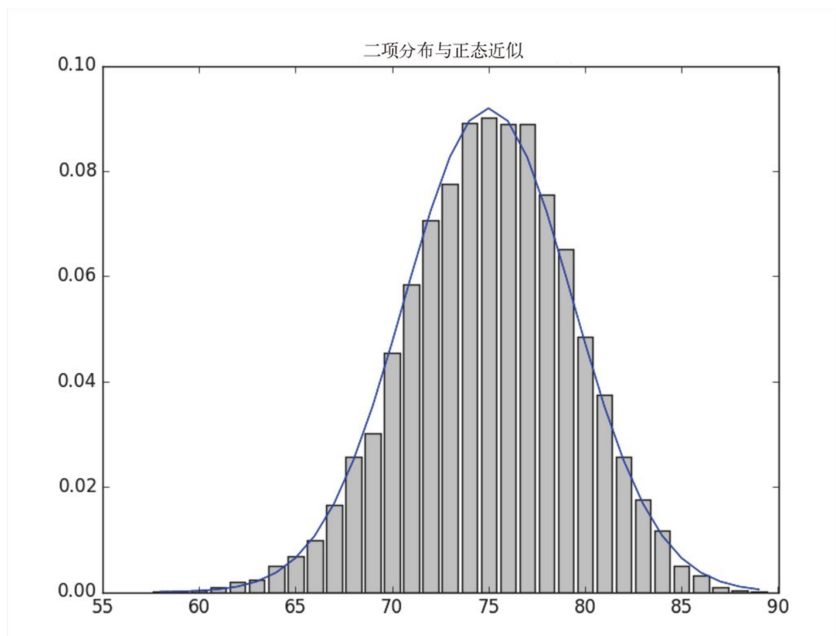


图 6-4：**`make_hist`** 的结果

近似表达的意义在于，如果你想知道掷起一枚均匀的硬币 100 次中正面朝上超过 60 次的概率，那么可以用一个正态分布 Normal(50, 5) 的随机变量大于 60 的概率来估计。这比计算二项式分布 Binonimal(100, 0.5) 的累积分布函数更容易（尽管在大多数应用中，你可以借助统计软件方便地计算出任何你想要的概率）。

6.8　延伸学习

- scipy.stats（http://docs.scipy.org/doc/scipy/reference/stats.html）包括绝大多数常见概率分 布的概率密度函数和累积分布函数。
- 在第 5 章末我曾建议读者学习统计学教材，同样，这里我也建议读者学习概率论教材。我所知道的最好的在线教材是 *Introduction to Probability*（http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/amsbook.mac.pdf）。

第 7 章　假设与推断

深谙统计之道，方为人中之龙。

——萧伯纳

具备以上统计学和概率理论知识以后，我们接着该做什么呢？数据科学的科学部分，乃是不断针对我们的数据和生成数据的机制建立假设和检验假设。

7.1　统计假设检验

作为数据科学家，我们常常需要检验某个假设是否成立。有时，假设是诸如“这枚硬币是均匀的”“数据科学家喜欢 Python 胜过 R”或“如果人们点开某个突然弹出的小广告，广告的关闭按钮又小又难找，那么大家更倾向于离开这个页面，压根不会阅读”等可以被翻译成统计数据的断言。在各种各样的假设之下，这些统计数据可以理解为从某种已知分布中抽取的随机变量观测值，这可以让我们对这些假设是否成立做出论断。

典型的步骤是这样的，首先我们有一个零假设 H_0，它代表一个默认的立场，而替代假设 H_1 代表我们希望与零假设对比的立场。我们通过统计来决定我们是否可以拒绝 H_0，即判断它是否错误。通过举例能更直观地说明这个过程。

7.2　案例：掷硬币

假设有一枚硬币，我们试图判断它是否均匀，即任何一面朝上的可能性是否相等。首先，假设硬币落地后正面朝上的概率为 p，所以我们的零假设为硬币均匀，即 $p=0.5$。我们要对

比替代假设 $p \neq 0.5$ 来检验这个假设。

具体来说，首先掷硬币 n 次，将出现正面朝上的次数记为 X。每次掷硬币都是一次伯努利试验，意味着 X 是二项式随机变量 Binomial(n,p)，（正如第 6 章中所讲到的）可以用正态分布来拟合：

```
def normal_approximation_to_binomial(n, p):
```

```
    """finds mu and sigma corresponding to a Binomial(n, p)"""
    mu = p * n
    sigma = math.sqrt(p * (1 - p) * n)
    return mu, sigma
```

只要一个随机变量服从正态分布，我们就可以用 normal_cdf 来计算出一个实现数值位于（或不在）某个特定区间的概率：

```
# 正态cdf是一个变量在一个阈值以下的概率
normal_probability_below = normal_cdf

# 如果它不在阈值以下，就在阈值之上
def normal_probability_above(lo, mu=0, sigma=1):
    return 1 - normal_cdf(lo, mu, sigma)

# 如果它小于hi但不比lo小，那么它在区间之内
def normal_probability_between(lo, hi, mu=0, sigma=1):
    return normal_cdf(hi, mu, sigma) - normal_cdf(lo, mu, sigma)

# 如果不在区间之内，那么就在区间之外
def normal_probability_outside(lo, hi, mu=0, sigma=1):
    return 1 - normal_probability_between(lo, hi, mu, sigma)
```

或者反过来，找出非尾区域，或者找出均值两边的（对称）区域，这个区域恰好对应特定比例的可能性。比如，如果我们需要找出以均值为中心、覆盖 60% 可能性的区间，那我们需要找到两个截点，使上尾和下尾各覆盖 20% 的可能性（给中间留出 60%）：

```
def normal_upper_bound(probability, mu=0, sigma=1):
    """returns the z for which P(Z <= z) = probability"""
    return inverse_normal_cdf(probability, mu, sigma)

def normal_lower_bound(probability, mu=0, sigma=1):
    """returns the z for which P(Z >= z) = probability"""
    return inverse_normal_cdf(1 - probability, mu, sigma)

def normal_two_sided_bounds(probability, mu=0, sigma=1):
    """returns the symmetric (about the mean) bounds
    that contain the specified probability"""
    tail_probability = (1 - probability) / 2

    # 上界应有在它之上的tail_probability
    upper_bound = normal_lower_bound(tail_probability, mu, sigma)

    # 下界应有在它之下的tail_probability
    lower_bound = normal_upper_bound(tail_probability, mu, sigma)

    return lower_bound, upper_bound
```

具体来讲，首先我们选择掷硬币 $n=1000$ 次。如果关于均匀的原假设是正确的，那么 X 近似服从正态分布，均值为 50，标准差为 15.8：

```
mu_0, sigma_0 = normal_approximation_to_binomial(1000, 0.5)
```

我们需要对显著性（significance）下定义——我们有多大的可能性犯第 **1** 类错误（“容错”）。在这种情况下，我们拒绝了原假设 H_0，但实际上原假设是正确的。出于历史上的某些原因，可能性的大小通常设定为 5% 或者 1%。本书在此选择 5%。

考虑这样的检验——如果 X 落在以下区间以外，就拒绝原假设 H_0：

```
normal_two_sided_bounds(0.95, mu_0, sigma_0) # (469, 531)
```

假设 p 实际上等于 0.5（即，此时 H_0 成立），那么我们有 5% 的可能观测到 X 落在区间之外，这正是我们想要的显著性。换句话说，如果 H_0 为真，那么 20 次检验中大约有 19 次会得出正确的结果。

我们常常对检验的势（power）有兴趣，它指的是不犯第 **2** 类错误的概率。第 2 类错误指原假设 H_0 是错的，但我们的检验结果没有拒绝原假设（即“纳伪”）。为了衡量统计的势，我们需要精确衡量 H_0 是错的意味着什么。（仅仅知道 p 不是 0.5 不足以为 X 的分布提供足够的信息。）具体来说，假如 p 实际上是 0.55，那么掷硬币的结果会稍微多偏向正面朝上。

在这种情形下，我们这样计算检验的势：

```
# 基于假设p是0.5时95%的边界
lo, hi = normal_two_sided_bounds(0.95, mu_0, sigma_0)

# 基于p = 0.55的真实mu和sigma
mu_1, sigma_1 = normal_approximation_to_binomial(1000, 0.55)

# 第2类错误意味着我们没有拒绝原假设
# 这会在X仍然在最初的区间时发生
type_2_probability = normal_probability_between(lo, hi, mu_1, sigma_1)
power = 1 - type_2_probability      # 0.887
```

如果我们把原假设变为掷硬币的结果不会偏重于正面朝上，即 $P \leq 0.5$，在这种情形下，我们使用单边检验。如果 X 远大于 50，我们就拒绝原假设，如果 X 小于 50，就不拒绝原假设。因此，显著性为 5% 的检验需要使用 normal_probability_below 来找出小于 95% 的 概率对应的截点：

```
hi = normal_upper_bound(0.95, mu_0, sigma_0)
# 是526 (< 531, 因为我们在上尾需要更多的概率)
type_2_probability = normal_probability_below(hi, mu_1, sigma_1)
power = 1 - type_2_probability      # 0.936
```

这是更有效的检验。如果 X 小于 469，我们就不再拒绝 H_0（如果 H_1 为真，这不太可能发生），当 X 在 526 和 531 之间时则拒绝 H_0（如果 H_1 为真，这很有可能发生）。

进行上述检验的另一种方式涉及 p 值。我们不再基于某个概率截点选择临界值，而是计算概率——假设 H_0 正确——我们可以找到一个至少与我们实际观测到的值一样极端的值。

对于硬币是否均匀的双面检验，我们可以做以下计算：

```
def two_sided_p_value(x, mu=0, sigma=1):
    if x >= mu:
        # 如果x大于均值，tail表示比x大多少
        return 2 * normal_probability_above(x, mu, sigma)
    else:
        # 如果x比均值小，tail表示比x小多少
        return 2 * normal_probability_below(x, mu, sigma)
```

如果我们希望看到结果中有 530 次为正面朝上，可以这样计算：

```
two_sided_p_value(529.5, mu_0, sigma_0)   # 0.062
```



为什么我们用 529.5 而不用 530 ？这就是所谓的连续校正（continuity correction）。它反映了一个事实，即对从掷硬币结果中看到 530 次正面朝上的概率而言，normal_probability_between(529.5, 530.5, mu_0,sigma_0) 是比

normal_probability_between(530, 531, mu_0, sigma_0) 更好的估计。

相应地，normal_probability_above(529.5, mu_0, sigma_0) 是看到至少 530 次正面朝上概率的更好估计。你可以在通过代码生成的图 6-4 中看到。

验证这种观点是否合理的一个方法是模拟：

```
extreme_value_count = 0
for _ in range(100000):
    num_heads = sum(1 if random.random() < 0.5 else 0     # 正面朝上的计数
                    for _ in range(1000))                 # 在1000次抛掷中
    if num_heads >= 530 or num_heads <= 470:              # 并计算达到极值的频
        extreme_value_count += 1                          # 极值的频率

print extreme_value_count / 100000   # 0.062
```

因为 p 值大于 5% 的显著性，所以我们不能拒绝原假设。如果我们看到了 532 次正面朝上，那么相应的 p 值为：

```
two_sided_p_value(531.5, mu_0, sigma_0) # 0.0463
```

它小于 5% 的显著性，因此我们拒绝原假设。它正好是和之前相同的检验，只是计算统计量的方法稍有不同。

同样，我们有：

```
upper_p_value = normal_probability_above
lower_p_value = normal_probability_below
```

对于单边检验，如果我们看到 525 次正面朝上，那么可以计算：

```
upper_p_value(524.5, mu_0, sigma_0)   # 0.061
```

这意味着我们会拒绝原假设。如果我们看到 527 次正面朝上，相应计算如下：

```
upper_p_value(526.5, mu_0, sigma_0)   # 0.047
```

```

```

根据结果，我们会拒绝原假设。



在调用函数 `normal_probability_above` 计算 p 值之前，需要确定你的数据大致上服从正态分布。数据科学的不良数据记录中充斥着差之毫厘失之千里的例子，原因在于“数据是正态分布的”，如果数据本身不是正态分布，那结果就毫无意义。

对正态分布的检验方法有好几种，绘图是不错的首选方案。

7.3　置信区间

我们一直在对正面朝上的概率 p 进行假设检验，这是未知的“正面朝上”分布的参数。对假设检验，我们还有第三种方法：在参数的观测值附近建立置信区间（confidence interval）。

例如，我们可以通过计算每次抛掷对应的伯努利随机变量的均值来估计不均匀硬币的概率——正面朝上记为 1，背面朝上记为 0，取这一系列伯努利随机变量的平均值。如果我们观测的 1000 次抛掷中有 525 次正面朝上，那么我们可以估计出 p 等于 0.525。

但是这个估计的可信度有多大呢？如果我们已知 p 的精确值，那么根据中心极限定理（见 6.7 节），伯努利随机变量的均值近似服从正态分布，其中均值为 p，标准为：

```
math.sqrt(p * (1 - p) / 1000)
```

这里，p 是未知的，所以我们使用估值：

```
p_hat = 525 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000)   # 0.0158
```

这种计算是不严格的，但运用广泛。借助正态近似我们得出结论：以

下区间包含真实参数 p 的可能性为 95%：

```
normal_two_sided_bounds(0.95, mu, sigma)        # [0.4940, 0.5560]
```

这是关于区间的解释，不是关于 p 值的解释。你需要这样理解：如果你重复实验很多次，其中 95% 的“真”参数（每次都相同）会落在观测到的置信区间（每次可能会都不同）中。

注意，我们没有得出不均匀硬币的结论，因为 0.5 落入了我们的置信区间。

如果我们观察到的是 540 次正面朝上，那么相应计算为：

```
p_hat = 540 / 1000
mu = p_hat
sigma = math.sqrt(p_hat * (1 - p_hat) / 1000) # 0.0158
normal_two_sided_bounds(0.95, mu, sigma) # [0.5091, 0.5709]
```

在这种情形下，“均匀硬币”没有落入置信区间。（“均匀硬币”的假设没有通过检验。如果假设是真的，那需要在 95% 的时间中都能通过。）

7.4　P-hacking

如果一个程序仅有 5% 的时间错误地拒绝了原假设，那么根据定义，5% 的时间会错误地拒绝原假设：

```
def run_experiment():
    """flip a fair coin 1000 times, True = heads, False = tails"""
    return [random.random() < 0.5 for _ in range(1000)]

def reject_fairness(experiment):
    """using the 5% significance levels"""
    num_heads = len([flip for flip in experiment if flip])
    return num_heads < 469 or num_heads > 531

random.seed(0)
experiments = [run_experiment() for _ in range(1000)]
num_rejections = len([experiment
                      for experiment in experiments
                      if reject_fairness(experiment)])
```

```
print num_rejections   # 46
```

这意味着如果你有意找出“显著”结果，那么总是可以的。只要对数据的假设检验次数足够多，就总有某些会表现出显著性。移除右边的那些异常值，你就可以得到小于 0.05 的 p 值。（注意，这与 5.2 节讲的相关性有些类似。）

这就是所谓的 P-hacking（http://www.nature.com/news/scientific-method-statistical-errors-1.14700），它某种程度上是“基于 p 值框架的推断”的结果。批评这种方法的一篇绝好文章是“地球是圆的”（http://ist-socrates.berkeley.edu/~maccoun/PP279_Cohen1.pdf）。

如果想做好科学工作，就要在审查数据之前确定你的假设，就要在做假设之前整理好你的数据，并且要牢记，p 值并不是靠直觉得出的。（一个替代方案是下文 7.6 节“贝叶斯推断”。）

7.5　案例：运行A/B测试

你在 DataSciencester 的主要职责之一是经验值优化，这是个委婉的说法，其实就是设法让人点击广告。你的一个广告商针对数据科学家开发了一种新的能量饮料，广告部门的副总希望你帮助他在广告 A（“口味好”）和广告 B（“营养均衡”）之间进行选择。

作为一名科学家，你得做一次实验，对网站访问者随机放送不同的广告，并记录每个广告的点击数。

如果 1000 个看到广告 A 的人中有 990 个人点击广告，而 1000 个看到广告 B 的人中只有 10 个点击，你可以确认 A 是更棒的广告。但倘若区别并不如此分明，可以使用统计推断进行选择。

假设有 N_A 个人看到广告 A，其中 n_A 个人点击广告。每次广告浏览都是一次伯努利试验，其中 p_A 是点击广告 A 的概率。然后（如果 N_A 足够大。此处就足够大）我们知道 n_A/N_A 是近似服从正态分布的随机变量，其中均值为 p_A，标准差为 $\sigma_A = \sqrt{p_A(1-p_A)/N_A}$。同样，$n_B/N_B$ 是近似服从正态分布的随机变量，均值为 p_B，标准差为 $\sigma_B = \sqrt{p_B(1-p_B)/N_B}$：

```
def estimated_parameters(N, n):
```

```
    p = n / N
    sigma = math.sqrt(p * (1 - p) / N)
    return p, sigma
```

如果我们假设这两个正态分布互相独立（这个假设是合理的，因为每个伯努利试验也是独立的），那么它们的差也是正态分布的，其中均值为 p_B - p_A，标准差为$\sqrt{\sigma_A^2 + \sigma_B^2}$。

 这某种程度上有些欺骗性。只有在标准差已知的条件下数学推理才正确。我们从数据中估计参数，这意味着我们实际中应该用 t 分布。但如果数据集足够大，正态分布和 t 分布之间的差别可以忽略不计。

这意味着我们可以检验 p_A 和 p_B 相等（即 p_A-p_B 等于零）这个原假设，具体方式如下：

```
def a_b_test_statistic(N_A, n_A, N_B, n_B):
    p_A, sigma_A = estimated_parameters(N_A, n_A)
    p_B, sigma_B = estimated_parameters(N_B, n_B)
    return (p_B - p_A) / math.sqrt(sigma_A ** 2 + sigma_B ** 2)
```

这应该近似一个标准正态分布。

比如，如果“口味好”的广告从 1000 次浏览中获得 200 次点击量，而“营养均衡”广告则从 1000 次浏览中获得 180 次点击量，则统计量等于：

```
z = a_b_test_statistic(1000, 200, 1000, 180)    # -1.14
```

如果两个均值实际上相等，那么看到如此大的差异的概率为：

```
two_sided_p_value(z)                            # 0.254
```

这计算出的数值很大，以至于你不可以得出有差距的结论。另一方面，如果“营养均衡”仅仅获得 150 次点击量，则：

```
z = a_b_test_statistic(1000, 200, 1000, 150)    # -2.94
two_sided_p_value(z)                            # 0.003
```

这意味着如果广告的效果相同，那么你看到有明显差异的概率只有 0.003。

7.6　贝叶斯推断

我们所看到的处理方式都包含对检验所做的与概率有关的陈述："如果原假设正确，那么你观测到极端统计量的概率仅有 3%。"

推断的一个替代方法是将未知参数视为随机变量。分析师（也就是你）从参数的先验分布（prior distribution）出发，再利用观测数据和贝叶斯定理计算出更新后的后验分布（posterior distribution）。不再对检验本身给出概率判断，而是对参数本身给出概率判断。

比如，如果未知参数是概率（就像掷硬币的例子），我们使用 **Beta** 分布（Beta distribution）作为先验分布，Beta 分布仅对 0 和 1 赋值：

```
def B(alpha, beta):
    """a normalizing constant so that the total probability is 1"""
    return math.gamma(alpha) * math.gamma(beta) / math.gamma(alpha + beta)

def beta_pdf(x, alpha, beta):
    if x < 0 or x > 1:          # [0, 1]之外没有权重
        return 0
    return x ** (alpha - 1) * (1 - x) ** (beta - 1) / B(alpha, beta)
```

一般来说，以上分布的权重中心为：

```
alpha / (alpha + beta)
```

`alpha` 和 `beta` 越大，分布就越"紧密"。

例如，如果 `alpha` 和 `beta` 都是 1，那么刚好是均匀分布（以 0.5 为中心，非常分散）。如果 `alpha` 比 `beta` 大很多，那么大多数权重接近 1。如果 `alpha` 比 `beta` 小很多，那么大多数权重接近零。图 7-1 展示了几种不同的 Beta 分布。

让我们先假设一个先验分布 p。如果对硬币是否均匀不预设立场，那么将 `alpha` 与 `beta` 都设定为 1。或者如果我们坚信硬币有 55% 的可能正面朝上，就选择让 `alpha` 等于 55，`beta` 等于 45。

然后我们多次掷起硬币，结果有 h 次正面朝上，有 t 次背面朝上。根据贝叶斯定理（和一些太过冗繁的数学，此处不赘述），p 的先验分布仍然是 Beta 分布，但参数分别为 `alpha + h` 和 `beta + t`。



后验分布也是 Beta 分布，这并非偶然。二项分布给出了正面朝上的数字，Beta 是二项分布的共轭先验分布（conjugate prior，http://www.johndcook.com/blog/conjugate_prior_diagram/）。这意味着，无论你何时使用从相关的二项分布中得到的观测值更新 Beta 先验分布，你还是会得到一个 Beta 后验分布。

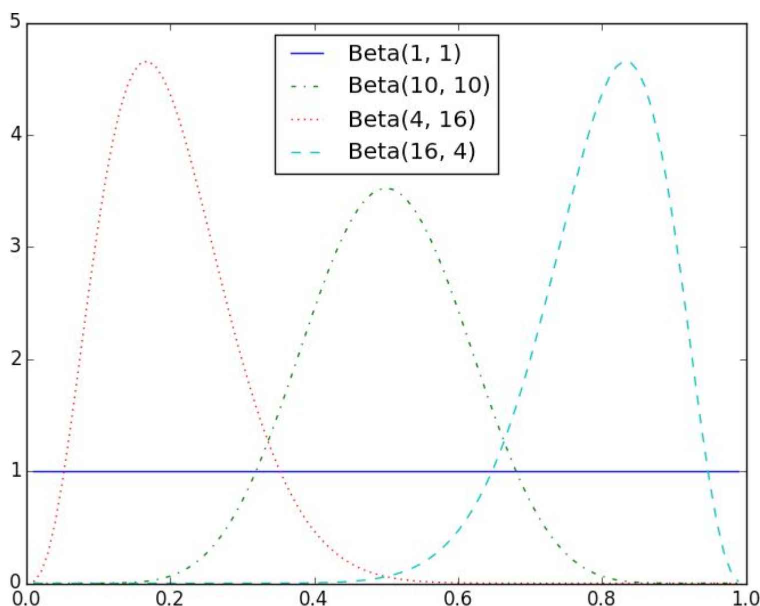


图 **7-1**：**Beta** 分布举例

假设你掷硬币 10 次并且观测到 3 次正面朝上。

如果你从均匀分布的先验开始（有时候不会采取硬币均匀的立场），那么你的后验分布为 Beta(4, 8)，中心为 0.33。如果你认为所有的可能性都相等，那么你最好的猜测就会非常接近观测到的概率。

如果你从 Beta(20, 20) 开始（表明硬币大致上是均匀的），那么你的后验分布为 Beta(23, 27)，中心为 0.46，这表明可能硬币稍稍倾向于背面朝上。

如果你从 Beta(30, 10) 开始（表明硬币是不均匀的，即有 75% 的可能会正面朝上），那么你的后验分布为 Beta(33, 17)，中心为 0.66。这种情况下，你仍然相信正面朝上的概率会大一些，只是没有一开始那么坚定了。这几个不同的后验分布如图 7-2 所示。

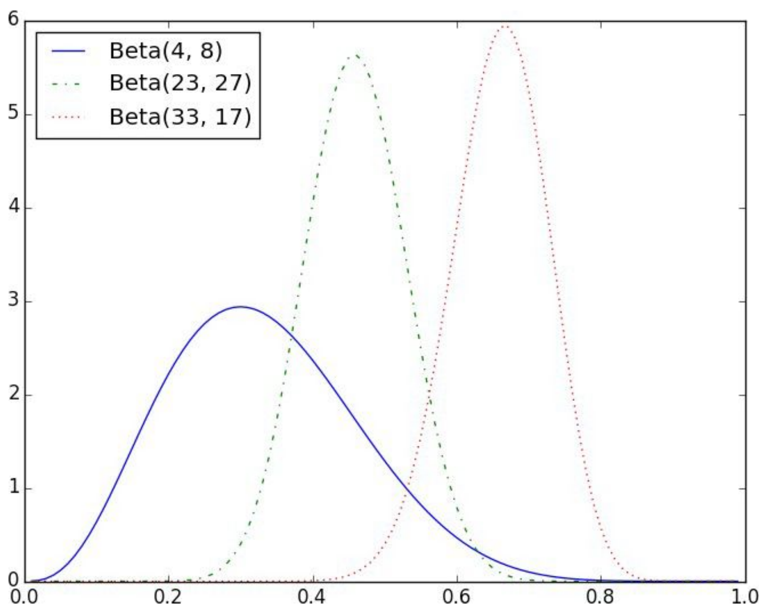


图 **7-2**：从不同先验分布得到的后验分布

如果你多次掷硬币，无论你最初选择了什么样的先验分布，先验分布对后验分布的影响会越来越小，直到最后得到（几乎）相同的后验分布。

比如，无论你最初对掷硬币的结果有怎样的倾向猜想，当看到 2000 次掷硬币的结果中有 1000 次正面朝上时，你都会很难维持原先的看法（除非你极端地选择了 Beta(1000000, 1) 这样的先验分布）。

有趣的是，这允许我们对假设"基于先验分布和已观测数据，正面朝上的概率介于 49% ~ 51% 的可能性仅有 5%"做出概率判断。这在哲学上不同于论断"如果硬币是均匀的，那么只有 5% 的机会能观测到极端数据"。

用贝叶斯推断进行假设检验是饱受争议的——部分源于它的数学原理非常复杂，部分源于选择先验分布的主观性。本书中我们不会挖掘得太深，但稍作了解还是有益的。

7.7 延伸学习

- 我们仅仅讲解了统计推断的一点皮毛知识。第 5 章末推荐的相关书籍有大量更加深入的细节知识。
- 在线公开课提供了涵盖许多相关主题的数据分析与统计推断课程（https://www.coursera.org/course/statistics）。

第 8 章　梯度下降

夸耀自己血统者，实际上夸耀的是他们对别人的亏欠。

——塞内加

从事数据科学工作时，常常会面临这样的需要：为某种特定的情形寻找最佳模型。“最佳”常常会被解读为某种类似于“最小化模型残差”或者“最大化数据的可能性”。换句话说，它代表了优化某种问题的解决方案。

这意味着我们需要解决一连串的最优化问题。尤其是，我们需要从零开始解决问题。我们采用的方法是一种叫作梯度下降（gradient descent）的技术，适合从零开始逐步解决问题。也许你无法从中体味出兴奋感，但它会在本书中教我们做很多令人兴奋的事情，所以，务必跟随我。

8.1　梯度下降的思想

假设我们拥有某个函数 `f`，这个函数输入一个实数向量，输出一个实数。一个简单的例子如下：

```
def sum_of_squares(v):
    """computes the sum of squared elements in v"""
    return sum(v_i ** 2 for v_i in v)
```

我们常常需要最大化（或最小化）这个函数。这意味着我们需要找出能计算出最大（或最小）可能值的输入 `v`。

对我们的函数来说，梯度（在微积分里，这表示偏导数向量）给出了输入值的方向，在这个方向上，函数增长得最快。（如果记不起微积分，用我提到的关键词上网查查。）

相应地，最大化函数的算法首先从一个随机初始点开始，计算梯度，在梯度方向（这是使函数增长最快的一个方向）上跨越一小步，再从一个新的初始点开始重复这个过程。同样，你也可以在相反方向上逐步最小化函数，如图 8-1 所示。

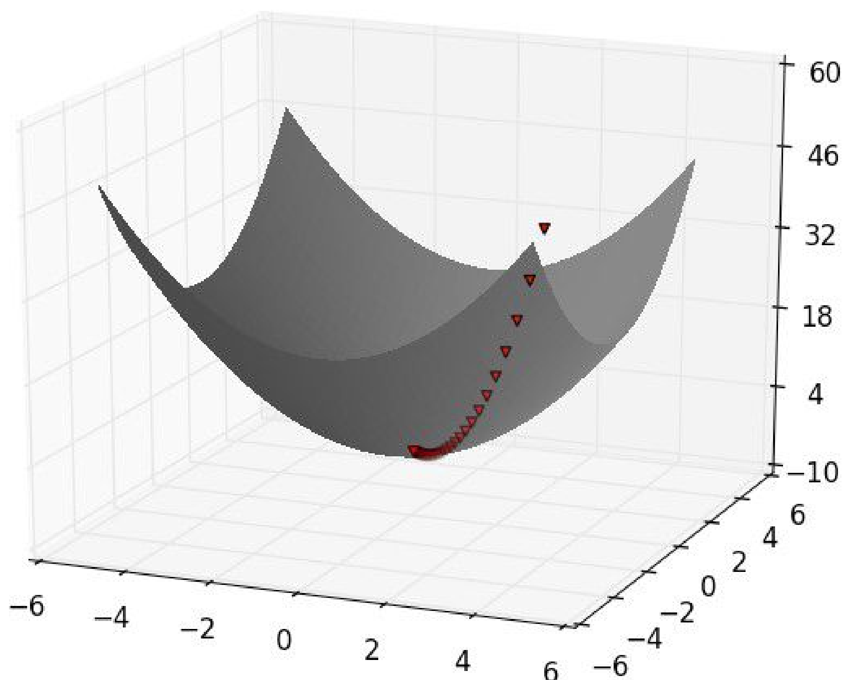


图 **8-1**：用梯度下降法计算最小点

 如果一个函数有一个全局最小点，那么这个方法很可能会找到它。如果这个函数有多个（局部）最小点，那么这种方法可能找不到这个点，但你可以通过多尝试一些初始点来重复运行这个方法。如果一个函数没有最小点，也许计算会陷入死循环。

8.2　估算梯度

如果 `f` 是单变量函数，那么它在 `x` 点的导数衡量了当 `x` 发生变化时，`f(x)` 变化了多少。导数通过差商的极限来定义：

```
def difference_quotient(f, x, h):
    return (f(x + h) - f(x)) / h
```

其中 `h` 趋近于 0。

（许多微积分初学者常常受困于极限的数学定义。这里我们不妨说，

你认为它是什么，它就是什么。）

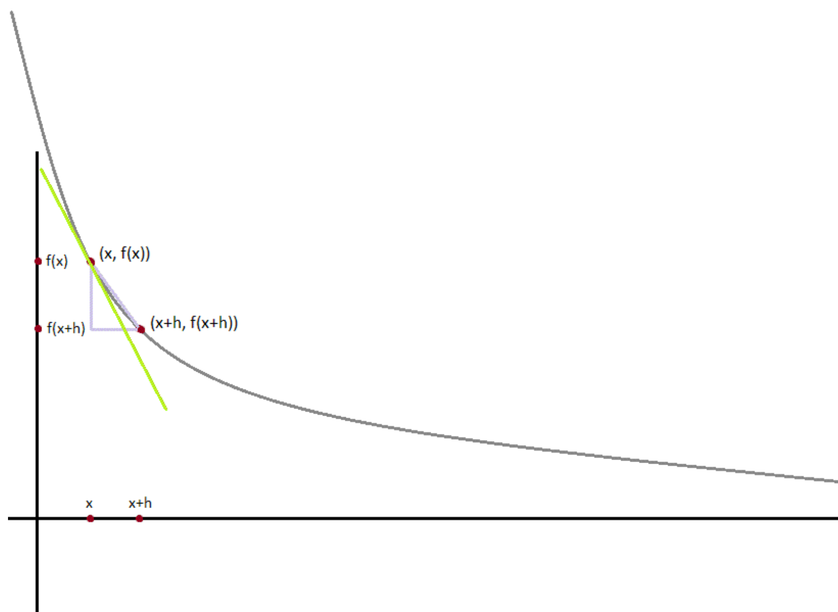


图 **8-2**：通过差商来求近似导数

导数就是在点 $(x, f(x))$ 的切线的斜率，而差商就是通过点 $(x, f(x))$ 和点 $(x+h, f(x+h))$ 的割 线的斜率。当 h 越来越小，割线与切线就越来越接近（见图 8-2）。

很多函数可以精确地计算导数，比如平方函数 `square`：

```
def square(x):
    return x * x
```

它的导数为：

```
def derivative(x):
    return 2 * x
```

你可以通过计算来确认——如果你想的话——先显式地计算差商，再取极限。

如果算不出梯度（或不想算）呢？ Python 中无法直接运算极限，但

可以通过计算一个很小的变动 e 的差商来估算微分。图 8-3 给出了这个估算的结果：

```
derivative_estimate = partial(difference_quotient, square, h=0.00001)

# 绘出导入matplotlib.pyplot作为plt的基本相同的形态
x = range(-10,10)
plt.title("精确的导数值与估计值")
plt.plot(x, map(derivative, x), 'rx', label='Actual')            # 用 x 表
plt.plot(x, map(derivative_estimate, x), 'b+', label='Estimate') # 用 + 表
plt.legend(loc=9)
plt.show()
```

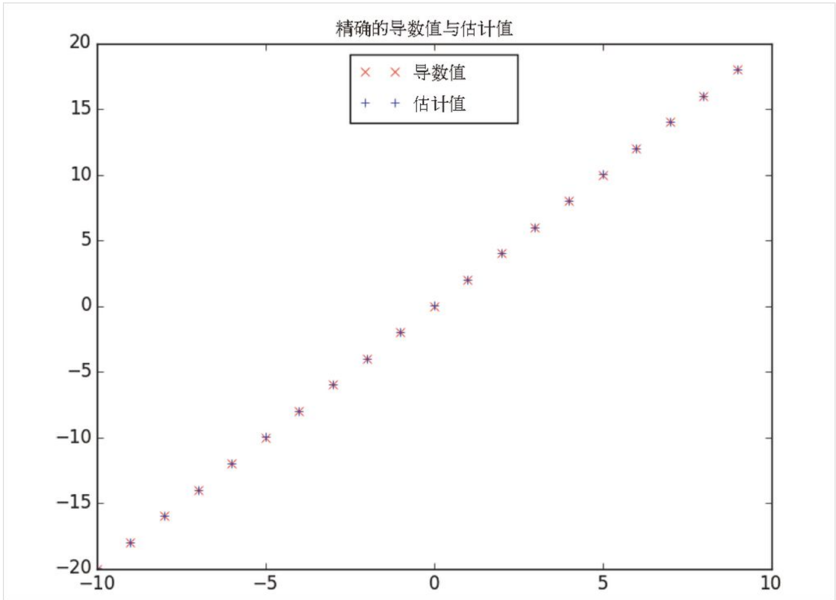


图 **8-3**：差商近似值的拟合度

当 f 是一个多变量函数时，它有多个偏导数，每个偏导数表示仅有一个输入变量发生微小变化时函数 f 的变化。

我们把导数看成是其第 *i* 个变量的函数，其他变量保持不变，以此来计算它第 *i* 个偏导数：

```
def partial_difference_quotient(f, v, i, h):
    """compute the ith partial difference quotient of f at v"""
    w = [v_j + (h if j == i else 0)     # 只对v的第i个元素增加h
         for j, v_j in enumerate(v)]
```

```
    return (f(w) - f(v)) / h
```

再以同样的方法估算它的梯度函数：

```
def estimate_gradient(f, v, h=0.00001):
    return [partial_difference_quotient(f, v, i, h)
            for i, _ in enumerate(v)]
```



“差商估算法”的主要缺点是计算代价很大。如果 v 长度为 n，那么 estimate_gradient 为了计算 f 需要 $2n$ 个不同的输入变量。如果你需要反复计算梯度，那需要做很多额外的工作。

8.3　使用梯度

很容易看出，当输入 v 是零向量时，函数 sum_of_squares 取值最小。但如果不知道输入是什么，可以用梯度方法从所有的三维向量中找到最小值。我们先找出随机初始点，并在梯度的反方向以小步逐步前进，直到梯度变得非常非常小：

```
def step(v, direction, step_size):
    """move step_size in the direction from v"""
    return [v_i + step_size * direction_i
            for v_i, direction_i in zip(v, direction)]

def sum_of_squares_gradient(v):
    return [2 * v_i for v_i in v]

# 选取一个随机初始值
v = [random.randint(-10,10) for i in range(3)]

tolerance = 0.0000001

while True:
    gradient = sum_of_squares_gradient(v)    # 计算v的梯度
    next_v = step(v, gradient, -0.01)        # 取负的梯度步长
    if distance(next_v, v) < tolerance:      # 如果收敛了就停止
        break
    v = next_v                               # 如果没汇合就继续
```

如果运行以上程序，你会发现，它总是止于一个非常接近 [0,0,0] 的 v 值。tolerance 值设定得越小，v 值就越接近 [0,0,0]。

8.4 选择正确步长

尽管向梯度的反向移动的逻辑已经清楚了，但移动多少还不明了。事实上，选择合适的步长更像艺术而非科学。主流的选择方法有：

- 使用固定步长
- 随时间增长逐步减小步长
- 在每一步中通过最小化目标函数的值来选择合适的步长

最后一种方法看上去不错，但它的计算代价也最大。我们可以尝试一系列步长，并选出使目标函数值最小的那个步长来求其近似值：

```
step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]
```

某些步长可能会导致函数的输入无效。所以，我们需要创建一个对无效输入值返回无限值（即这个值永远不会成为任何函数的最小值）的“安全应用”函数：

```
def safe(f):
    """return a new function that's the same as f,
    except that it outputs infinity whenever f produces an error"""
    def safe_f(*args, **kwargs):
        try:
            return f(*args, **kwargs)
        except:
            return float('inf')         # 意思是Python中的“无限值”
    return safe_f
```

8.5 综合

通常而言，我们有一些 target_fn 函数，需要对其进行最小化，也有梯度函数 gradient_fn。比如，函数 target_fn 可能代表模型的残差，它是参数的函数。我们可能需要找到能使残差尽可能小的参数。

此外，假设我们（以某种方式）为参数 theta_0 设定了某个初始值，那么可以如下使用梯度下降法：

```
def minimize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
    """use gradient descent to find theta that minimizes target function"""

    step_sizes = [100, 10, 1, 0.1, 0.01, 0.001, 0.0001, 0.00001]

    theta = theta_0                              # 设定theta为初始值
    target_fn = safe(target_fn)                  # target_fn的安全版
    value = target_fn(theta)                     # 我们试图最小化的值

    while True:
        gradient = gradient_fn(theta)
        next_thetas = [step(theta, gradient, -step_size)
                       for step_size in step_sizes]

    # 选择一个使残差函数最小的值
    next_theta = min(next_thetas, key=target_fn)
    next_value = target_fn(next_theta)

    # 当"收敛"时停止
    if abs(value - next_value) < tolerance:
        return theta
    else:
        theta, value = next_theta, next_value
```

我们称它为 minimize_batch，因为在每一步梯度计算中，它都会搜索整个数据集（因为 target_fn 代表整个数据集的残差）。在下一部分中，我们会探讨另一种方法，一次仅考虑一个数据点。

有时候，我们需要最大化某个函数，这只需要最小化这个函数的负值（相应的梯度函数也需取负）：

```
def negate(f):
    """return a function that for any input x returns -f(x)"""
    return lambda *args, **kwargs: -f(*args, **kwargs)

def negate_all(f):
    """the same when f returns a list of numbers"""
    return lambda *args, **kwargs: [-y for y in f(*args, **kwargs)]

def maximize_batch(target_fn, gradient_fn, theta_0, tolerance=0.000001):
    return minimize_batch(negate(target_fn),
                          negate_all(gradient_fn),
                          theta_0,
                          tolerance)
```

8.6　随机梯度下降法

正如之前提到的，我们常常用梯度下降的方法，通过最小化某种形式的残差来选择模型参数。如果使用之前的批处理方法，每个梯度计算步都需要我们预测并计算整个数据集的梯度，这使每一步都会耗费很长时间。

现在，这些残差函数常常具有可加性（additive），意味着整个数据集上的预测残差恰好是每个数据点的预测残差之和。

在这种情形下，我们转而使用一种称为随机梯度下降（stochastic gradient descent）的技术，它每次仅计算一个点的梯度（并向前跨一步）。这个计算会反复循环，直到达到一个停止点。

在每个循环中，我们都会在整个数据集上按照一个随机序列迭代：

```
def in_random_order(data):
    """generator that returns the elements of data in random order"""
    indexes = [i for i, _ in enumerate(data)]  # 生成索引列表
    random.shuffle(indexes)                    # 随机打乱数据
    for i in indexes:                          # 返回序列中的数据
        yield data[i]
```

我们对每个数据点都会进行一步梯度计算。这种方法留有这样一种可能性，即也许会在最小值附近一直循环下去，所以，每当停止获得改进，我们都会减小步长并最终退出：

```
def minimize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.

    data = zip(x, y)
    theta = theta_0                             # 初始值猜测
    alpha = alpha_0                             # 初始步长
    min_theta, min_value = None, float("inf")   # 迄今为止的最小值
    iterations_with_no_improvement = 0

    # 如果循环超过100次仍无改进，停止
    while iterations_with_no_improvement < 100:
        value = sum( target_fn(x_i, y_i, theta) for x_i, y_i in data )

        if value < min_value:
            # 如果找到新的最小值，记住它
            # 并返回到最初的步长
            min_theta, min_value = theta, value
            iterations_with_no_improvement = 0
            alpha = alpha_0
        else:
            # 尝试缩小步长，否则没有改进
            iterations_with_no_improvement += 1
```

```
        alpha *= 0.9

    # 在每个数据点上向梯度方向前进一步
    for x_i, y_i in in_random_order(data):
        gradient_i = gradient_fn(x_i, y_i, theta)
        theta = vector_subtract(theta, scalar_multiply(alpha, gradient

return min_theta
```

随机化通常比批处理化快很多。当然，我们也希望获得最大化的结果：

```
def maximize_stochastic(target_fn, gradient_fn, x, y, theta_0, alpha_0=0.0
    return minimize_stochastic(negate(target_fn),
                               negate_all(gradient_fn),
                               x, y, theta_0, alpha_0)
```

8.7 延伸学习

- 继续往下读，我们将用梯度下降法解决本书剩余部分提到的很多问题。
- 如果继续推荐你阅读教材，你该感到厌倦了。幸好，这次推荐你读的是 *Active Calculus*（http://scholarworks.gvsu.edu/books/10/），它似乎比我读过的任何微积分教材都要友好一些。
- scikit-learn 有一个随机梯度下降模块（http://scikit-learn.org/stable/modules/sgd.html），它在某些方面讲得比较笼统，而在其他一些方面讲得很详细。事实上，在真实世界的大多数场景中，你都会用到库，库中的优化技术已经充分考虑到背后的原理了，所以你不必为此操心（绝不会某个时候就不能正常工作了，毫无疑问，这不可能）。

第 9 章　获取数据

写作本书我用了三个月的时间；构思只用了三分钟；而收集书中的数据，则用了我的一生。

——F. 斯科特 · 菲兹杰拉德

为了成为一名数据科学家，你需要数据。事实上，作为数据科学家，你会花超大一部分时间来获取、清理和转换数据。必要时，你总可以自己输入数据（或者可以让你的助手来做），但通常这样做比较浪费时间。本章我们来看看利用 Python 获取数据并得到正确格式的不同方法。

9.1　stdin和stdout

如果在命令行运行 Python 脚本，你可以用 `sys.stdin` 和 `sys.stdout` 以管道（pipe）方式传递数据。例如，以下脚本按行读入文本，然后划分出和一个正则表达式匹配的行：

```
# egrep.py
import sys, re

# sys.argv是命令行参数的列表
# sys.argv[0]是程序自己的名字
# sys.argv[1]会是在命令行上指定的正则表达式
regex = sys.argv[1]

# 对传递到这个脚本中的每一个行
for line in sys.stdin:
    # 如果它匹配正则表达式，则把它写入stdout
    if re.search(regex, line):
        sys.stdout.write(line)
```

然后对收到的行计数并输出计数结果：

```
# line_count.py
import sys

count = 0
for line in sys.stdin:
    count += 1
```

```
# 输出去向 sys.stdout
print count
```

你可以用这种方法来计数文件中有多少行包含数字。在 Windows 中，你可以用：

```
type SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

而在 Unix 系统中，你可以用：

```
cat SomeFile.txt | python egrep.py "[0-9]" | python line_count.py
```

“|”运算符是个管道字符，它的意思是“使用左边命令的输出作为右边命令的输入”。可以使用这种方法精心设计数据处理的管道。



如果你使用的是 Windows，你可以在这行命令中去掉所包含的 python 部分：

```
type SomeFile.txt | egrep.py "[0-9]" | line_count.py
```

如果你用 Unix 系统，这么做可能会需要多做一些额外工作。

类似地，下面的这个脚本计算了单词的数量并给出了最常用的单词：

```
# most_common_words.py
import sys
from collections import Counter

# 传递单词的个数作为第一个参数
try:
    num_words = int(sys.argv[1])
except:
    print "usage: most_common_words.py num_words"
    sys.exit(1)    # 非零的exit代码表明有错误

counter = Counter(word.lower()                          # 小写的单词
                  for line in sys.stdin
                  for word in line.strip().split()  # 用空格划分
```

```
                  if word)                            # 跳过空的 'words'

for word, count in counter.most_common(num_words):
    sys.stdout.write(str(count))
    sys.stdout.write("\t")
    sys.stdout.write(word)
    sys.stdout.write("\n")
```

之后你可以对圣经文本使用这个脚本：

```
C:\DataScience>type the_bible.txt | python most_common_words.py 10
64193   the
51380   and
34753   of
13643   to
12799   that
12560   in
10263   he
9840    shall
8987    unto
8836    for
```

如果你是个 Unix 编程老手，你可能会很熟悉系统里许多内置的命令行工具（比如 egrep），但你自己从零开始创建这些工具或许更好一些。毕竟，用到的时候刚好知道，总是好的。

9.2 读取文件

可以显式地用代码来读写文件。用 Python 处理文件非常简便。

9.2.1 文本文件基础

处理文本文件的第一步是通过 open 命令来获取一个文件对象：

```
# 'r' 意味着只读
file_for_reading = open('reading_file.txt', 'r')

# 'w' 是写入——会破坏已存在的文件！
file_for_writing = open('writing_file.txt', 'w')

# 'a' 是添加——加入到文件的末尾
file_for_appending = open('appending_file.txt', 'a')
```

```
# 完成以后别忘了关闭文件
file_for_writing.close()
```

因为非常容易忘记关闭文件，所以你应该在 with 程序块里操作文件，这样在结尾处文件会被自动关闭：

```
with open(filename,'r') as f:
    data = function_that_gets_data_from(f)

# 此时，f已经关闭了，别再试图使用它
process(data)
```

如果需要读取一个完整的文本文件，可以使用 for 语句对文件的行进行迭代：

```
starts_with_hash = 0

with open('input.txt','r') as f:
    for line in file:                  # 查找文件中的每一行
        if re.match("^#",line):        # 用正则表达式判断它是否以'#'开头
            starts_with_hash += 1      # 如果是，计数加1
```

按这种方法得到的每一行会用换行符来结尾，所以在对读入的行操作之前会经常需要用 strip() 来进行处理。

例如，假设你有一个写满电子邮件地址的文件，每个地址一行，你想利用这个文件生成域名的直方图。正确地提取域名的规则有些微妙（如公共后缀列表，https://publicsuffix.org/），但一个好的近似方案是只取出电子邮件地址中 @ 后面的部分。（对于像 joel@mail.datasciencester.com 这样的邮件地址，会给出错误的答案。）

```
def get_domain(email_address):
    """split on '@' and return the last piece"""
    return email_address.lower().split("@")[-1]

with open('email_addresses.txt', 'r') as f:
    domain_counts = Counter(get_domain(line.strip())
                            for line in f
                            if "@" in line)
```

9.2.2 限制的文件

我们刚刚处理的假想电子邮件地址文件每行只有一个地址。更常见的情况是你会处理每一行包含许多数据的文件。这种文件通常是用逗号分割或 **tab** 分割的，每一行有许多字段，用逗号（或 tab）来表示一个字段的结束和另一个字段的开始。

这开始变得复杂了，各字段中带有逗号、tab 和换行符（这是你不可避免地要处理的）。因为这个原因，几乎总是会犯的一个错误是你自己尝试去解析它们。相反，你应该使用 Python 的 `csv` 模块（或者 `pandas` 库）。出于微软方面（你可以对其大加责备）的技术原因，你应该总是通过把 *b* 包括在 *r* 或 *w* 之后来用二进制模式处理 `csv` 文件（见 Stack Overflow，http://stackoverflow.com/questions/4249185/using-python-to-append-csv-files）。

如果文件没有头部（意味着你可能想把每一行作为一个列表，这带来的麻烦是你需要知道每一列是什么），你可以使用 `csv.reader` 对行进行迭代，每一行都会被处理成恰当划分的列表。

例如，如果有这样一个用 tab 划分的股票价格文件：

```
6/20/2014	AAPL	90.91
6/20/2014	MSFT	41.68
6/20/2014	FB	64.5
6/19/2014	AAPL	91.86
6/19/2014	MSFT	41.51
6/19/2014	FB	64.34
```

我们可以用下面的程序块来处理：

```
import csv

with open('tab_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.reader(f, delimiter='\t')
    for row in reader:
        date = row[0]
        symbol = row[1]
        closing_price = float(row[2])
        process(date, symbol, closing_price)
```

如果文件存在头部：

```
date:symbol:closing_price
6/20/2014:AAPL:90.91
6/20/2014:MSFT:41.68
6/20/2014:FB:64.5
```

你既可以跳过头部的行（利用对 read.next() 的初始调用）也可以利用 csv.DictReader 把每一行读成字典（把头部作为关键字）：

```
with open('colon_delimited_stock_prices.txt', 'rb') as f:
    reader = csv.DictReader(f, delimiter=':')
    for row in reader:
        date = row["date"]
        symbol = row["symbol"]
        closing_price = float(row["closing_price"])
        process(date, symbol, closing_price)
```

即使你的文件缺少头部，你仍可以通过把关键字作为文件名参数传输来使用 DictReader。

同样，你可以用 csv.writer 来写限制的文件：

```
today_prices = { 'AAPL' : 90.91, 'MSFT' : 41.68, 'FB' : 64.5 }

with open('comma_delimited_stock_prices.txt','wb') as f:
    writer = csv.writer(f, delimiter=',')
    for stock, price in today_prices.items():
        writer.writerow([stock, price])
```

如果行中的各字段本身包含逗号，csv.writer 可以正确处理。但你自己手动写成的则很可能不会正确处理。比如，如果你尝试这样做：

```
results = [["test1", "success", "Monday"],
           ["test2", "success, kind of", "Tuesday"],
           ["test3", "failure, kind of", "Wednesday"],
           ["test4", "failure, utter", "Thursday"]]

# 不要这么做！
with open('bad_csv.txt', 'wb') as f:
    for row in results:
        f.write(",".join(map(str, row))) # 可能包含了太多逗号！
        f.write("\n")                    # 行也可能会换行！
```

你最终会得到像下面这样一个 csv 文件：

```
test1,success,Monday
test2,success, kind of,Tuesday
test3,failure, kind of,Wednesday
test4,failure, utter,Thursday
```

没人能看懂它的意思。

9.3 网络抓取

另一种获取数据的方法是从网页抓取数据。获取一个网页十分容易，但从网页上抓取有意义的结构化信息就不那么容易了。

9.3.1 HTML和解析方法

网络上的页面是由 HTML 写成的，其中文本被（理想化地）标记为元素和它们的属性：

```
<html>
  <head>
    <title>A web page</title>
  </head>
  <body>
    <p id="author">Joel Grus</p>
    <p id="subject">Data Science</p>
  </body>
</html>
```

在理想的情况下，所有的网页为我们方便地按语义标记，我们可以使用类似这样的规则来提取数据：找到 id 是 subject 的 <p> 元素并返回它所包含的文本。但在真实的世界中，HTML 并不总是具有很好的格式的，更不用说注解了。这意味着如果我们想搞清其含义，需要一些帮助。

为了从 HTML 里得到数据，我们需要使用 BeatifulSoup 库（http://www.crummy.com/software/BeautifulSoup/），它对来自网页的多种元素建立了树结构，并提供了简单的接口来获取它们。本书写作时，最新的版本是 Beatiful Soup 4.3.2（pip install beautifulsoup4），我们即将用到的就是这个版本。我们也会用到 requests 库（pip install requests，http://docs.python-

requests.org/en/latest/ ），它与内置在 Python 中的其他方法相比，是一种发起 HTTP 请求的更好的方式。

Python 内置的 HTML 解析器是有点严格的，这意味着它并不总是能处理那些没有很好地格式化的 HTML。因此，我们需要使用另外一种解析器，它需要先安装：

```
pip install html5lib
```

为了使用 Beatiful Soup，我们要把一些 HTML 传递给 `BeautifulSoup()` 函数。在我们的例子中，这些 HTML 是对 `requests.get` 进行调用的结果：

```
from bs4 import BeautifulSoup
import requests
html = requests.get("http://www.example.com").text
soup = BeautifulSoup(html, 'html5lib')
```

完成这个步骤之后，我们可以用一些简单的方法得到完美的解析。

通常我们会处理一些 `Tag` 对象，它们对应于 HTML 页面结构的标签表示。

比如，找到你能用的第一个 `<p>` 标签（及其内容）：

```
first_paragraph = soup.find('p')        # 或仅仅soup.p
```

可以对 `Tag` 使用它的 `text` 属性来得到文本内容：

```
first_paragraph_text = soup.p.text
first_paragraph_words = soup.p.text.split()
```

另外可以把标签当作字典来提取其属性：

```
first_paragraph_id = soup.p['id']        # 如果没有'id'则报出KeyError
first_paragraph_id2 = soup.p.get('id')    # 如果没有'id'则返回None
```

可以一次得到多个标签：

```
all_paragraphs = soup.find_all('p')      # 或仅仅soup('p')
paragraphs_with_ids = [p for p in soup('p') if p.get('id')]
```

通常你会想通过一个类（class）来找到标签：

```
important_paragraphs = soup('p', {'class' : 'important'})
important_paragraphs2 = soup('p', 'important')
important_paragraphs3 = [p for p in soup('p')
                         if 'important' in p.get('class', [])]
```

此外，可以把这些方法组合起来运用更复杂的逻辑。比如，如果想找出包含在一个 <div> 元素中的每一个 <span> 元素，可以这么做：

```
# 警告，将多次返回同一个span元素
# 如果它位于多个div元素里
# 如果是这种情况，要更谨慎一些
spans_inside_divs = [span
                     for div in soup('div')      # 对页面上的每个<div>
                     for span in div('span')]    # 找到其中的每一个<span>
```

仅仅上述几个特性就可以帮助我们做很多事。如果你需要做更复杂的事情（或仅仅是出于好奇），那就去查看文档吧。

当然，无论多重要的数据，通常也不会标记成 class="important"。你需要仔细检查源 HTML，通过你选择的逻辑进行推理，并多考虑边界情况来确保数据的正确性。接下来我们看一个例子。

9.3.2　案例：关于数据的O'Reilly图书

DataSciencester 的某位潜在投资者认为数据只会风靡一时。为了证明他是错的，你打算查看一下 O'Reilly 出版社这些年来总共出版过多少数据类的图书。通过对 O'Reilly 网站的挖掘，你发现它有许多有关数据图书（以及视频）的页面，每次 30 个条目的目录页面有这样的 URL：

```
http://shop.oreilly.com/category/browse-subjects/data.do?sortby=publicatio
```

我们都不笨（而且也不想让自己的抓取器被封），所以在每次从网站抓取数据之前都该看一下这家网站是否有某种获取政策。查看以下页面：

```
http://oreilly.com/terms/
```

看起来对这个项目没有明文禁止。但为了做一个守法的好公民，我们还应该查看一下 robots.txt 文件，看看一个网络抓取者要有怎样的行为规范。http://shop.oreilly.com/robots.txt 有以下重要内容：

```
Crawl-delay: 30
Request-rate: 1/30
```

第一行告诉我们应该在两次请求之间等待 30 秒，第二行告诉我们每 30 秒只能请求一个页面。所以从根本上说这两行原则讲的是同一件事。（文件里还有一些内容说明有些目录页是不能抓取的，但是我们的 URL 不在其中，所以我们可以放心了。）



O'Reilly 是有可能改变某些网站政策的，那样会打破本小节的所有逻辑。我会尽我所能预防这种情况的发生，当然，我对 O'Reilly 并没有太大的影响力。然而，如果你们每人都发动所有认识的人买一本这书的话……

为了弄清该怎样提取数据，让我们下载其中一个页面，把它传给 Beatiful Soup：

```
# 除非是写进书里，否则你没必要这样拆分一个url
url = "http://shop.oreilly.com/category/browse-subjects/" + \
      "data.do?sortby=publicationDate&page=1"
soup = BeautifulSoup(requests.get(url).text, 'html5lib')
```

如果你查看页面的源代码（在浏览器中右键选择"查看源代码"或"查看网页源代码"，或其他最接近的选项），会看到每本书（或每部视频）都唯一地包含在一个表格单元格元素 `<td>` 中，它的类是 `thumbtext`。下面的内容是某本书相关的 HTML（一个删减的版本）：

```
<td class="thumbtext">
  <div class="thumbcontainer">
```

```
    <div class="thumbdiv">
      <a href="/product/9781118903407.do">
        <img src="..."/>
      </a>
    </div>
  </div>
  <div class="widthchange">
    <div class="thumbheader">
      <a href="/product/9781118903407.do">Getting a Big Data Job For Dummi
    </div>
    <div class="AuthorName">By Jason Williamson</div>
    <span class="directorydate">        December 2014     </span>
    <div style="clear:both;">
      <div id="146350">
        <span class="pricelabel">
                            Ebook:

                            <span class="price"> $29.99</span>
        </span>
      </div>
    </div>
  </div>
</td>
```

良好的开端是找到所有的 td thumbtext 标签元素：

```
tds = soup('td', 'thumbtext')
print len(tds)
# 30
```

接下来我们要过滤掉视频。（那位潜在的投资者只对书感兴趣。）如果我们进一步地检查 HTML，会看到每个 td 会包含一个或更多个类为 pricelabel 的 span 元素，它的文本看起来像 Ebook: 或者 video: 或者 Print:。看起来视频仅包含一个 pricelabel，它的文本以 Video（在移除前导空格之后）开头。这意味着我们可以这样来检测视频：

```
def is_video(td):
    """it's a video if it has exactly one pricelabel, and if
    the stripped text inside that pricelabel starts with 'Video'"""
    pricelabels = td('span', 'pricelabel')
    return (len(pricelabels) == 1 and
            pricelabels[0].text.strip().startswith("Video"))

print len([td for td in tds if not is_video(td)])
# 对我来说结果是21，你得到的结果可能会不同
```

现在我们已准备好要从 td 元素中提取数据了。看起来图书的标题是包含在 <div class="thumbheader"> 里的标签 <a> 中的文本：

```
title = td.find("div", "thumbheader").a.text
```

作者（们）的名字在 AuthorName <div> 的文本里。它们由一个 By（我们打算去掉它）开头，由逗号分隔（我们打算把它们分隔开，然后去掉其中的空格）：

```
author_name = td.find('div', 'AuthorName').text
authors = [x.strip() for x in re.sub("^By ", "", author_name).split(",")]
```

ISBN 看起来是包含在 thumbheader <div> 中的链接里：

```
isbn_link = td.find("div", "thumbheader").a.get("href")

# re.match捕捉了括号中的正则表达式部分
isbn = re.match("/product/(.*)\.do", isbn_link).group(1)
```

日期就是 <span class="directorydate"> 的内容：

```
date = td.find("span", "directorydate").text.strip()
```

让我们把所有这些都放到一个函数里边：

```
def book_info(td):
    """given a BeautifulSoup <td> Tag representing a book,
    extract the book's details and return a dict"""

    title = td.find("div", "thumbheader").a.text
    by_author = td.find('div', 'AuthorName').text
    authors = [x.strip() for x in re.sub("^By ", "", by_author).split(",")]
    isbn_link = td.find("div", "thumbheader").a.get("href")
    isbn = re.match("/product/(.*)\.do", isbn_link).groups()[0]
    date = td.find("span", "directorydate").text.strip()

    return {
        "title" : title,
        "authors" : authors,
        "isbn" : isbn,
        "date" : date
```

```
    }
```

现在我们准备好进行抓取了：

```
from bs4 import BeautifulSoup
import requests
from time import sleep
base_url = "http://shop.oreilly.com/category/browse-subjects/" + \
           "data.do?sortby=publicationDate&page="

books = []

NUM_PAGES = 31     # 这是写作本书时的值，现在有可能更多

for page_num in range(1, NUM_PAGES + 1):
    print "souping page", page_num, ",", len(books), " found so far"
    url = base_url + str(page_num)
    soup = BeautifulSoup(requests.get(url).text, 'html5lib')

    for td in soup('td', 'thumbtext'):
        if not is_video(td):
            books.append(book_info(td))

# 现在做一个好公民，遵守robots.txt！
sleep(30)
```

像这样从 HTML 中提取数据更像是一种数据艺术而不是数据科学。除了上例之外，你还可以从 HTML 中实施查找图书、查找标题等不计其数的类似的逻辑行为。

既然已经收集好了数据，现在就可以把每一年出版的图书数据绘制出来（如图 9-1）：

```
def get_year(book):
    """book["date"] looks like 'November 2014' so we need to
    split on the space and then take the second piece"""
    return int(book["date"].split()[1])

# 2014是包含数据的最后一个完整的年份（我运行这段代码的时间）
year_counts = Counter(get_year(book) for book in books
                      if get_year(book) <= 2014)

import matplotlib.pyplot as plt
years = sorted(year_counts)
book_counts = [year_counts[year] for year in years]
plt.plot(years, book_counts)
```

```
plt.ylabel("数据图书的数量")
plt.title("数据大发展！")
plt.show()
```

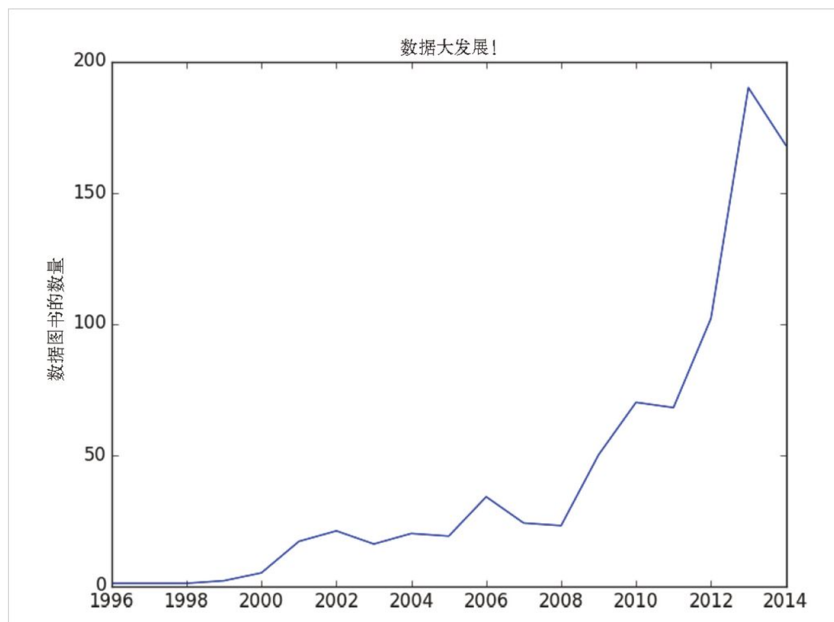


图 **9-1**：每年数据图书的出版数量

不幸的是，那位潜在的投资者看到这张图后断言 2013 年是“数据时代的巅峰”。

9.4　使用API

许多网站和网络服务提供相应的应用程序接口（Application Programming Interface，API），允许你明确地请求结构化格式的数据。这省去了你不得不抓取数据的麻烦！

9.4.1　JSON（和XML）

因为 HTTP 是一种转换文本的协议，你通过网络 API 请求的数据需要串行化（serialized）地转换为字符串格式。通常这种串行化使用 JavaScript 对象符号（JavaScript Object Notation，JSON）。JavaScript 对象看起来和 Python 的字典很像，使得字符串表达非常

容易解释：

```
{ "title" : "Data Science Book",
  "author" : "Joel Grus",
  "publicationYear" : 2014,
  "topics" : [ "data", "science", "data science"] }
```

我们可以使用 Python 的 `json` 模块来解析 JSON。尤其是，我们会用到它的 `loads` 函数，这个函数可以把一个代表 JSON 对象的字符串反串行化（deserialize）为 Python 对象：

```
import json
serialized = """{ "title" : "Data Science Book",
                  "author" : "Joel Grus",
                  "publicationYear" : 2014,
                  "topics" : [ "data", "science", "data science"] }"""

# 解析JSON以创建一个Python字典
deserialized = json.loads(serialized)
if "data science" in deserialized["topics"]:
    print deserialized
```

有时候 API 的提供者可能会不耐烦，只给你提供 XML 格式的响应：

```
<Book>
  <Title>Data Science Book</Title>
  <Author>Joel Grus</Author>
  <PublicationYear>2014</PublicationYear>
  <Topics>
    <Topic>data</Topic>
    <Topic>science</Topic>
    <Topic>data science</Topic>
  </Topics>
</Book>
```

我们也可以仿照从 HTML 获取数据的方式，用 `BeautifulSoup` 从 XML 中获取数据；更多细节可查阅文档。

9.4.2 使用无验证的API

现在大多数的 API 要求你在使用之前先验证身份。而若我们不愿勉强自己屈就这种政策，API 会给出许多其他的陈词滥调来阻止我们的浏览。因此，先来看一下 GitHub 的 API（https://

developer.github.com/v3/），利用它我们可以做一些简单的无需验证的事情：

```
import requests, json
endpoint = "https://api.github.com/users/joelgrus/repos"

repos = json.loads(requests.get(endpoint).text)
```

此处 `repos` 是一个 Python 字典的列表，其中每一个字典表示我的 GitHub 账户的一个代码仓库。（可以随意替换成你的用户名，以取得你的代码仓库的数据。你有 GitHub 账号，对吧？）

这个结果能指出哪个月哪一周的哪一天我最愿意创建代码仓库。唯一的问题是，响应里的日期是（Unicode）字符串：

```
u'created_at': u'2013-07-05T02:02:28Z'
```

Python 本身没有很强大的日期解析器，所以我们需要安装一个：

```
pip install python-dateutil
```

其中你需要的可能只是 `dateutil.parser.parse` 函数：

```
from dateutil.parser import parse

dates = [parse(repo["created_at"]) for repo in repos]
month_counts = Counter(date.month for date in dates)
weekday_counts = Counter(date.weekday() for date in dates)
```

类似地，你可以获取我最后五个代码仓库所用的语言：

```
last_5_repositories = sorted(repos,
                             key=lambda r: r["created_at"],
                             reverse=True)[:5]

last_5_languages = [repo["language"]
                    for repo in last_5_repositories]
```

通常我们无需在“做出请求而且自己解析响应”这种低层次上使用 API。使用 Python 的好处之一是已经有人建好了库，方便你访问你感

兴趣的几乎所有 API。这些库可以把事情做好，为你省下查找 API 访问的诸多冗长细节的麻烦。（如果这些库不能很好地完成任务，或者它们依赖的是对应的 API 已失效的版本，那就会给你带来巨大的麻烦。）

尽管如此，偶尔你还是需要操作你自己的 API 访问库（或者，更常见的，去调试别人不能顺利操作的库），所以了解一些细节是很有好处的。

9.4.3　寻找API

如果你需要一个特定网站的数据，可以查看它的开发者部分或 API 部分的细节，然后以关键词“python api”在网络上搜索相应的库。Python 有一个 Rotten Tomatoes 的库。Python 还有针对 Klout、Yelp、IMDB 等的多个 API 封装。

如果你想查看有 Python 封装的 API 列表，可参阅 Python API（http://www.pythonapi.com/）和 Python for Beginners（http://www.pythonforbeginners.com/development/list-of-python-apis/）中的两个名录。

如果你想要的是一份更宽泛的网络 API 名录（不一定含有 Python 封装），Programmable Web（http://www.programmableweb.com/）是个好的资源，它有一个关于分好类的 API 的庞大名录。

如果最终还是找不到你需要的 API，还是可以通过抓取获得的。这是数据科学家最后的绝招。

9.5　案例：使用Twitter API

Twitter 是一个非常好的数据源。你可以从它得到实时的新闻，可以用它来度量对当前事件的反应，可以利用它找到与特定主题有关的链接。使用 Twitter 可以做几乎任何你能想到的事，只要你能获得它的数据。可以通过它的 API 来获得数据。

为了和 Twitter API 互动，我们需要使用 Twython 库（`pip install twython`，https://github.com/ryanmcgrath/twython）。实际上有很多 Python Twitter 的库，但这一个是我用过的库中最好用的一个。你也可以尝试一下其他的库。

获取证明文件

为了使用 Twitter 的 API，需要先获取一些证明文件（为此你无论如何都要有一个 Twitter 的账户，这样你就能成为一个活跃友好的 Twitter `#datascience` 社区的一部分）。就像那些所有我不能控制的网站的指令一样，它们会在某个时刻过时，但是现在还是能发挥一段时间的作用的。（尽管在我写作本书的这段时间里，它们至少已经变更过一次了，所以祝你好运！）

1. 找到链接 https://apps.twitter.com/。

2. 如果你还没有注册，点击“注册”，并输入你的 Twitter 用户名和密码。

3. 点击“创建新 App”。

4. 给它起个名字（比如“数据科学”）并添加一些描述，放上一个任意的 URL 作为网址（不用在乎是哪个）。

5. 同意“服务条款”并点击“创建”。

6. 注意消费者钥匙（consumer key）和消费者密码（consumer secret）。

7. 点击“创建我的访问令牌”（access token）。

8. 注意访问令牌和访问令牌密码（你可能需要刷新页面）。

消费者钥匙和消费者密码告诉 Twitter 什么应用正在访问它的 API，而访问令牌和访问令牌密码告诉 Twitter 是谁正在访问它的 API。如果你曾经用 Twitter 账户访问过一些其他网站，“点击验证”页面会生成一个访问令牌，网站会用这个令牌来告诉 Twitter 访问者是你（或者至少是来自你的操作）。因为不需要这种“让任何人登录”的功能，我们可以获得静态生成的访问令牌和访问令牌密码。



消费者钥匙 / 密码和访问令牌钥匙 / 密码应该被看成是密码（password）。你不该分享它们，不该把它们印在书里，也不应该把它们记录在 GitHub 公共代码库里。一种简单的方法是把它们存储在不会被签入的 credentials.json 文件里，而且可以使用 `json.loads` 取回它们。

使用Twython

首先我们来看看 Search API（https://dev.twitter.com/docs/api/1.1/get/search/tweets），这个操作只需要消费者钥匙和密码，无需访问令牌或密码：

```
from twython import Twython

twitter = Twython(CONSUMER_KEY, CONSUMER_SECRET)

# 搜索包含短语"数据科学"的推文
for status in twitter.search(q='"data science"')["statuses"]:
    user = status["user"]["screen_name"].encode('utf-8')
    text = status["text"].encode('utf-8')
    print user, ":", text
    print
```



因为推文中经常包含 `print` 函数无法处理的 Unicode 字符，所以有必要使用 `.encode("utf-8")` 来应对这个问题。（如果对这个问题放任不管，很有可能会得到 `UnicodeEncodeError` 的报错。）

几乎可以肯定，你会在数据科学家职业生涯的某些时刻遇到严重的 Unicode 问题，这时你需要参考 Python 文档（https://docs.python.org/2/howto/unicode.html）或者勉为其难地开始使用 Python 3 吧，因为它能更好地处理 Unicode 文本。

如果你运行这段代码，会得到一些像这样的推文：

```
haithemnyc: Data scientists with the technical savvy &amp; analytical chop
derive meaning from big data are in demand. http://t.co/HsF9Q0dShP

RPubsRecent: Data Science http://t.co/6hcHUz2PHM

spleonard1: Using #dplyr in #R to work through a procrastinated assignment
@rdpeng in @coursera data science specialization.  So easy and Awesome.
```

这并不十分有趣，很大程度上是因为 Twitter Search API 只给你显示它认为最近的结果，无论内容有多么少。但对于数据科学工作，通常

你需要大量的推文。这时，Streaming API（https://dev.twitter.com/streaming/reference/get/statuses/sample）就有用武之地了，它允许你连接到强大的 Twitter firehose 接口（的一个样本）。你需要使用访问令牌进行验证，才可以使用这个 API。

为了用 Twython 访问 Streaming API，需要定义一个从 `TwythonStreamer` 继承的类，并用这个类的 `on_success` 方法覆盖（当然也可能是用它的 `on_error` 方法来覆盖）：

```
from twython import TwythonStreamer

# 把数据添加到全局变量是一种非常差的形式
# 但会让这个例子更简单
tweets = []

class MyStreamer(TwythonStreamer):
    """our own subclass of TwythonStreamer that specifies
    how to interact with the stream"""

    def on_success(self, data):
        """what do we do when twitter sends us data?
        here data will be a Python dict representing a tweet"""

        # 只收集英文的推文
        if data['lang'] == 'en':
            tweets.append(data)
            print "received tweet #", len(tweets)

        # 当收集了足够多的推文就停止
        if len(tweets) >= 1000:
            self.disconnect()

    def on_error(self, status_code, data):
        print status_code, data
        self.disconnect()
```

MyStreamer 会连接到 Twitter 流并等待 Twitter 给它发送数据。它每收到一些数据（在这里，一条推文表示为一个 Python 对象）就传递给 `on_success` 方法，如果推文是英文的，这个方法会把推文附加到 `tweets` 列表中，在收集到 1000 条推文后会断开和流的连接。

剩下的工作就是初始化和启动运行了：

```
stream = MyStreamer(CONSUMER_KEY, CONSUMER_SECRET,
                    ACCESS_TOKEN, ACCESS_TOKEN_SECRET)

# 开始使用包含关键词'data'的公共状态
```

```
stream.statuses.filter(track='data')

# 如果我们想使用*all*公共状态的样本
# stream.statuses.sample()
```

它会一直运行下去直到收集 1000 条推文为止（或直到遇到一个错误为止），此时就可以着手分析这些推文了。比如，你可以用下面的方法寻找最常见的标签：

```
top_hashtags = Counter(hashtag['text'].lower()
                       for tweet in tweets
                       for hashtag in tweet["entities"]["hashtags"])

print top_hashtags.most_common(5)
```

每条推文都包含许多数据。你可以自己尝试一下各种方法，或仔细查阅 Twitter API 的文档（https://dev.twitter.com/overview/api/tweets）。


在一个正式的项目中，你可能并不想依赖内存中的列表来存储推文。相反，你可能想把推文保存在文件或者数据库中，这样就可以永久地拥有它们。

9.6　延伸学习

- pandas（http://pandas.pydata.org/）是数据科学用来处理（特别是导入）数据的一个主要的库。
- Scrapy（http://scrapy.org/）是一个特性更全的库，可用来构建更复杂的网络抓取器，来执行类似跟踪未知链接等任务。

第 10 章　数据工作

专家更依赖数据，而非主观判断。

——科林 · 鲍威尔

数据工作既是艺术又是科学。前面我们讨论的大多是数据的科学的一面，这一章我们来管窥其艺术的一面。

10.1　探索你的数据

当确定了需要研究的问题，并已获取了一些数据时，你摩拳擦掌地恨不得马上建模求解。但是，你需要克制一下。首先，你应该探索数据。

10.1.1　探索一维数据

最简单的情形是，你得到的一个数据集合仅仅是一维数据集。比如，它们可以是每个用户在你的网站上平均每天花费的时间，每个数据科学教程视频的观看次数，或者是你的数据科学图书馆中每本数据科学书的页数。

第一步显然是计算一些总结性统计数据。比如你可能想知道你的数据集中有多少个数据点，最小值是多少，最大值是多少，平均值是多少，或者标准差是多少。

如果你仍不能很好地理解以上步骤，那么下一步最好是绘出直方图，即将你的数据分组成离散的区间（bucket），并对落入每个区间的数据点进行计数：

```
def bucketize(point, bucket_size):
    """floor the point to the next lower multiple of bucket_size"""
    return bucket_size * math.floor(point / bucket_size)

def make_histogram(points, bucket_size):
    """buckets the points and counts how many in each bucket"""
    return Counter(bucketize(point, bucket_size) for point in points)

def plot_histogram(points, bucket_size, title=""):
    histogram = make_histogram(points, bucket_size)
```

```
    plt.bar(histogram.keys(), histogram.values(), width=bucket_size)
    plt.title(title)
    plt.show()
```

比如，考虑以下两个数据集：

```
random.seed(0)

# -100到100之间均匀抽取
uniform = [200 * random.random() - 100 for _ in range(10000)]

# 均值为0标准差为57的正态分布
normal = [57 * inverse_normal_cdf(random.random())
          for _ in range(10000)]
```

这两个数据集的均值都接近 0，标准差都接近 58 ，但它们的分布非常不同。图 10-1 展示了均匀分布。

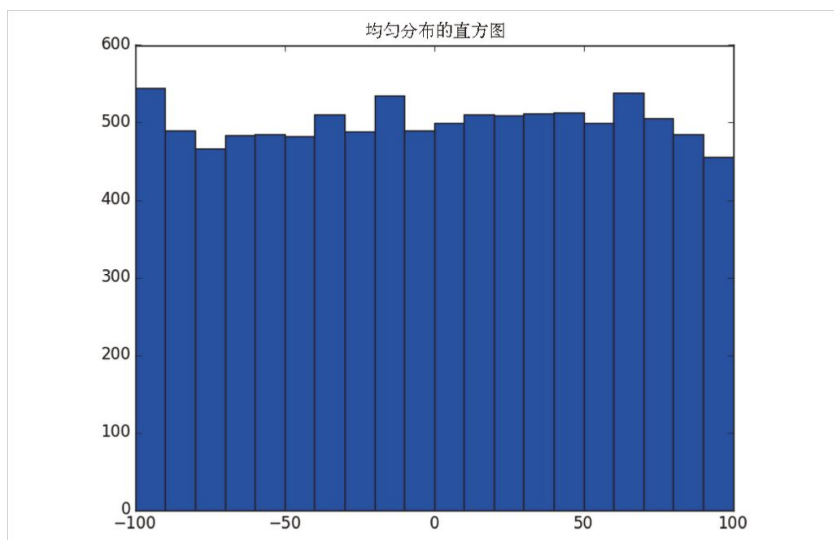


图 **10-1**：均匀分布的直方图

```
plot_histogram(uniform, 10, "均匀分布的直方图")
```

而图 10-2 展示了正态分布：

```
plot_histogram(normal, 10, "正态分布的直方图")
```

```

```

这两种分布有非常不同的最大值和最小值。但是，仅仅知道这一点并不足以理解它们有何差异。

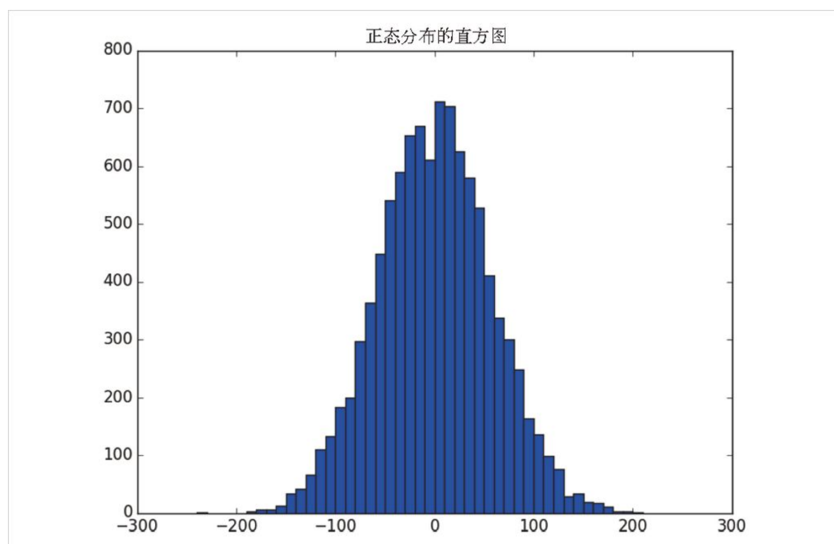


图 **10-2**：正态分布的直方图

10.1.2　二维数据

现在假设你的数据集是二维的。也许在每天上网时间之外还增加了数据科学工作年限。你当然会希望能从每个维度上单独理解数据，但也许你更希望综合两个维度来考察数据。

比如，考察下面一个伪数据集：

```
def random_normal():
    """returns a random draw from a standard normal distribution"""
    return inverse_normal_cdf(random.random())

xs = [random_normal() for _ in range(1000)]
ys1 = [ x + random_normal() / 2 for x in xs]
ys2 = [-x + random_normal() / 2 for x in xs]
```

如果你对 `ys1` 和 `ys2` 运行 `plot_histogram` 程序，会得到很相似的直方图（事实上，两个正态分布的均值和标准差都相同）。

但是在联合分布上，每个都与 `xs` 有很大差别，如图 10-3 所示：

```
plt.scatter(xs, ys1, marker='.', color='black', label='ys1')
plt.scatter(xs, ys2, marker='.', color='gray', label='ys2')
plt.xlabel('xs')
plt.ylabel('ys')
plt.legend(loc=9)
plt.title("差别很大的联合分布")
plt.show()
```

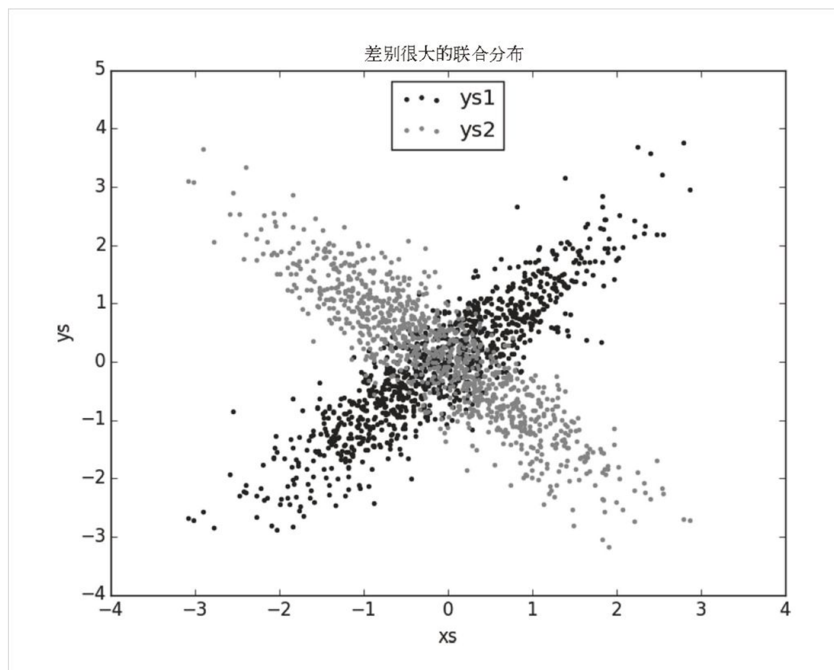


图 **10-3**：两个不同的 **ys** 的散点图

如果你考察相关性，差异会非常明显：

```
print correlation(xs, ys1)    # 0.9
print correlation(xs, ys2)    # -0.9
```

10.1.3　多维数据

对于多维数据，你可能想了解各个维度之间是如何相关的。一个简单的方法是考察相关矩阵（correlation matrix），矩阵中第 i 行第 j 列

的元素表示第 i 维与第 j 维数据的相关性：

```
def correlation_matrix(data):
    """returns the num_columns x num_columns matrix whose (i, j)th entry
    is the correlation between columns i and j of data"""

    _, num_columns = shape(data)

   def matrix_entry(i, j):
       return correlation(get_column(data, i), get_column(data, j))

   return make_matrix(num_columns, num_columns, matrix_entry)
```

一个更为直观的方法（如果维度不太多）是做散点图矩阵（图 10-4），以展示配对散点图。通过命令 `plt.subplots()` 可以生成子图。我们给出了行数和列数，它返回一个 `figure` 对象（我们不会用到它）和一个 `axes` 对象的二维数组（每个都会绘出）：

```
import matplotlib.pyplot as plt

_, num_columns = shape(data)
fig, ax = plt.subplots(num_columns, num_columns)

for i in range(num_columns):
    for j in range(num_columns):

        # x轴上column_j对y轴上column_i的散点
        if i != j: ax[i][j].scatter(get_column(data, j), get_column(data,

        # 只有当 i == j时显示序列名
        else: ax[i][j].annotate("series " + str(i), (0.5, 0.5),
                                xycoords='axes fraction',
                                ha="center", va="center")

        # 除了图的左侧和下方之外，隐藏图的标记
        if i < num_columns - 1: ax[i][j].xaxis.set_visible(False)
        if j > 0: ax[i][j].yaxis.set_visible(False)

# 修复右下方和左上方的图标记
# 因为它们只有文本，是错误的
ax[-1][-1].set_xlim(ax[0][-1].get_xlim())
ax[0][0].set_ylim(ax[0][1].get_ylim())

plt.show()
```

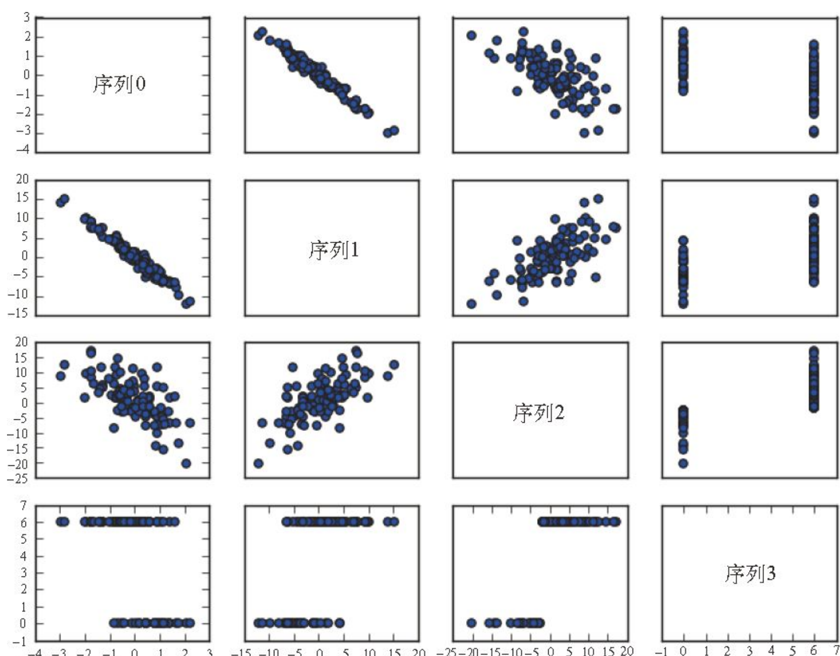


图 **10-4**：散点图矩阵

通过这些散点图你会看出，序列 1 与序列 0 的负相关程度很高，序列 2 和序列 1 的正相关程度很高，序列 3 的值仅有 0 和 6，并且 0 对应序列 2 中较小的值，6 对应较大的值。

这是一种能让你查看变量之间大概的相关度的快捷方法（除非你为了查看更加具体的效果而花费数小时调整 `matplotlib`，这样就不快捷了）。

10.2　清理与修改

真实世界的数据是有很多问题的。在使用数据之前，你通常需要对它们进行一定的预处理。我们在第 9 章举过这样的例子。我们需要把字符串转化成可以使用的浮点型数据（`float`）或者整型数据（`int`）。以前，我们在使用数据之前会这样做：

```
closing_price = float(row[2])
```

但通过建立包括 `csv.reader` 的函数，这样进行解析更不容易触发

误差。我们会列出一系列解析器，每个解析器具体说明其中一列如何解析。我们会用 None 表示“对这列什么都不做”：

```
def parse_row(input_row, parsers):
    """given a list of parsers (some of which may be None)
    apply the appropriate one to each element of the input_row"""

    return [parser(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]

def parse_rows_with(reader, parsers):
    """wrap a reader to apply the parsers to each of its rows"""
    for row in reader:
        yield parse_row(row, parsers)
```

如果有不良数据怎么办？一个浮点值是否真正代表一个数字？我们通常会使用一个 None 函数而非硬跑程序。我们可以通过一个辅助函数来解决：

```
def try_or_none(f):
    """wraps f to return None if f raises an exception
    assumes f takes only one input"""
    def f_or_none(x):
        try: return f(x)
        except: return None
    return f_or_none
```

然后我们重写 parse_row 来使用它：

```
def parse_row(input_row, parsers):
    return [try_or_none(parser)(value) if parser is not None else value
            for value, parser in zip(input_row, parsers)]
```

比如，如果我们用逗号分割的股票数据中有不良数据：

```
6/20/2014,AAPL,90.91
6/20/2014,MSFT,41.68
6/20/3014,FB,64.5
6/19/2014,AAPL,91.86
6/19/2014,MSFT,n/a
6/19/2014,FB,64.34
```

我们现在可以在一个单独步骤中读入和解析：

```
import dateutil.parser
data = []

with open("comma_delimited_stock_prices.csv", "rb") as f:
    reader = csv.reader(f)
    for line in parse_rows_with(reader, [dateutil.parser.parse, None, floa
        data.append(line)
```

然后我们只需检查其中 None 的行数：

```
for row in data:
    if any(x is None for x in row):
        print row
```

然后再决定如何处理它们。（一般来说，你有三个选择：删除它们；溯源并修复不良数据或缺失数据；什么都不做，自求多福吧。）

我们可以为 csv.DictReader 创建相似的帮助函数。这样的话，你很可能希望提供基于域名的解析字典。例如：

```
def try_parse_field(field_name, value, parser_dict):
    """try to parse value using the appropriate function from parser_dict"
    parser = parser_dict.get(field_name)    # 如果没有此条目，则为None
    if parser is not None:
        return try_or_none(parser)(value)
    else:
        return value

def parse_dict(input_dict, parser_dict):
    return { field_name : try_parse_field(field_name, value, parser_dict)
             for field_name, value in input_dict.iteritems() }
```

接下来最好是使用 10.1 节所讲的技术或即时分析来确认异常值。比如，如果你发现股票文件中有一个数据的时间是 3014 年，这不会给你报错提示，但这显然是错误的数据。如果你没有发现这个错误，就会得到很糟糕的结果。真实世界的数据集充斥着诸如小数点缺失、多余的零、排印错误等无数各种各样的错误，找出错误是你责无旁贷的工作。（也许这不是你的正式工作，但这工作又非你做不可。）

10.3　数据处理

数据科学家的核心技能之一就是处理数据。与其说它是一种特定的技

术，不如说它是一种通用的方法，所以这里我们只通过一些例子窥其一二。

假设我们需要处理如下股票价格字典：

```
data = [
    {'closing_price': 102.06,
     'date': datetime.datetime(2014, 8, 29, 0, 0),
     'symbol': 'AAPL'},
    # ...
]
```

我们可以从概念上将它们理解为行（就像在一张表中）。

我们开始对这些数据发问。在这个过程中，我们会不断关注做事所使用的模式，并抽象出一些工具以使数据的处理更容易些。

比如，如果我们想知道 AAPL 有史以来的最高收盘价，可以将这个工作分解成具体的步骤：

(1) 将数据限定在 AAPL 行上；

(2) 从每行提取收盘价 `closing_price`；

(3) 取价格中的最大值 `max`。

我们可以使用一个列表解析一次性完成这三个步骤：

```
max_aapl_price = max(row["closing_price"]
                     for row in data
                     if row["symbol"] == "AAPL")
```

更一般地，我们也许希望知道数据集中每只股票的最高收盘价。一个方法如下所示。

(1) 聚集起股票代码（`symbol`）相同的行。

(2) 在每组中，重复之前的工作：

```
# 按股票代码对行分组
by_symbol = defaultdict(list)
for row in data:
    by_symbol[row["symbol"]].append(row)
```

```
# 使用字典解析找到每个股票代码的最大值
max_price_by_symbol = { symbol : max(row["closing_price"]
                                     for row in grouped_rows)
                        for symbol, grouped_rows in by_symbol.iteritems()
```

有一些模式已经存在。在以上两个例子中，我们需要在每个字典 dict 中提取出收盘价 closing_price。因而我们可以创建一个函数，以从字典中提取一个字段，并创建另一个函数，以从字典集合中提取出同样的字段：

```
def picker(field_name):
    """returns a function that picks a field out of a dict"""
    return lambda row: row[field_name]

def pluck(field_name, rows):
    """turn a list of dicts into the list of field_name values"""
    return map(picker(field_name), rows)
```

我们同样可以建立一个函数，通过 group 函数的结果把行分组，并选择性地对每组使用 value_transform 函数：

```
def group_by(grouper, rows, value_transform=None):
    # 键是分组情况的输出，值是行的列表
    grouped = defaultdict(list)
    for row in rows:
        grouped[grouper(row)].append(row)

    if value_transform is None:
        return grouped
    else:
        return { key : value_transform(rows)
                 for key, rows in grouped.iteritems() }
```

这使得我们可以更简单地再现先前的例子。比如：

```
max_price_by_symbol = group_by(picker("symbol"),
                               data,
                               lambda rows: max(pluck("closing_price", row
```

现在我们可以问一些更复杂的问题，比如在我们的数据集中，单日百分比变动的最大值和最小值分别是什么。百分比变动的公式是 price_today/price_yesterday - 1（即今天的价格 / 昨天的

价格 -1 ）。这意味着我们需要用某种方式将今天的价格和昨天的价格联系起来。一种方法是按照符号将价格分组，再在每组中：

(1) 按照日期排列价格；

(2) 通过命令 `zip` 得到配对价格（前一天的，今天的）；

(3) 将配对价格转换为新的“百分比变动”行。

我们首先写一个函数，来完成每一组内的工作：

```
def percent_price_change(yesterday, today):
    return today["closing_price"] / yesterday["closing_price"] - 1

def day_over_day_changes(grouped_rows):
    # 按日期对行排序
    ordered = sorted(grouped_rows, key=picker("date"))

    # 对偏移量应用zip函数得到连续两天的成对表示
    return [{ "symbol" : today["symbol"],
              "date" : today["date"],
              "change" : percent_price_change(yesterday, today) }
            for yesterday, today in zip(ordered, ordered[1:])]
```

然后我们可以将它作为 `value_transform` 在 `group_by` 中使用：

```
# 键是股票代码，值是一个"change"字典的列表
changes_by_symbol = group_by(picker("symbol"), data, day_over_day_changes)

# 收集所有"change"字典放入一个大列表中
all_changes = [change
               for changes in changes_by_symbol.values()
               for change in changes]
```

在这个点上，很容易找到最大值与最小值：

```
max(all_changes, key=picker("change"))
# {'change': 0.3283582089552237,
#  'date': datetime.datetime(1997, 8, 6, 0, 0),
#  'symbol': 'AAPL'}
# see, e.g. http://news.cnet.com/2100-1001-202143.html
min(all_changes, key=picker("change"))
# {'change': -0.5193370165745856,
#  'date': datetime.datetime(2000, 9, 29, 0, 0),
#  'symbol': 'AAPL'}
```

```
# see, e.g. http://money.cnn.com/2000/09/29/markets/techwrap/
```

现在我们可以使用这个新的 all_changes 数据集来找出投资科技股的最佳月份。首先按月份对变化分组；然后在每组中计算整体变化。

我们再次写一个恰当的 value_transform 函数，然后使用 group_by 函数：

```
# 为了组合百分比的变化，我们对每一项加1，把它们相乘，再减去1
# 比如，如果我们组合 +10% 和 -20%， 总体的改变是
#    (1 + 10%) * (1 - 20%) - 1 = 1.1 * .8 - 1 = -12%
def combine_pct_changes(pct_change1, pct_change2):
    return (1 + pct_change1) * (1 + pct_change2) - 1

def overall_change(changes):
    return reduce(combine_pct_changes, pluck("change", changes))

overall_change_by_month = group_by(lambda row: row['date'].month,
                                   all_changes,
                                   overall_change)
```

类似这样的数据处理方式将会贯穿全书，但它常常不会明显地引起我们的注意。

10.4　数据调整

许多技术对数据单位（scale）敏感。比如，假设你有一个包括数百名数据科学家的身高和体重的数据集，并且需要创建体型大小的聚类（cluster）。

直观上讲，我们用聚类表示相近的点，这意味着我们需要某种点距离的概念。我们知道有欧几里得距离函数 distance，所以自然地，一种方法是将数据对 (height, weight) 视为二维空间中的点。考虑表 10-1 中列出的观测对象。

表**10-1**：身高和体重

	[illegible] 身高 [illegible] 体重 [illegible]		
[illegible] 150			
[illegible] 170.2			
[illegible] 177.8			

如果我们用英寸作为身高的单位，那么 B 最近的邻居是 A：

```
a_to_b = distance([63, 150], [67, 160])        # 10.77
a_to_c = distance([63, 150], [70, 171])        # 22.14
b_to_c = distance([67, 160], [70, 171])        # 11.40
```

但是，如果用厘米作为单位，那么 B 最近的邻居变成了 C：

```
a_to_b = distance([160, 150], [170.2, 160])    # 14.28
a_to_c = distance([160, 150], [177.8, 171])    # 27.53
b_to_c = distance([170.2, 160], [177.8, 171]) # 13.37
```

显然，如果单位变化导致结果发生这样的变化，那肯定是有问题的。因此，如果不同的维度之间不可比较，就需要对数据进行调整（rescale），以使得每个维度的均值为 0，标准差为 1。这种转换有效地摆脱了单位带来的问题，将每个维度转化为“与均值的标准差”。

首先，我们需要对每列计算均值和标准差：

```
def scale(data_matrix):
    """returns the means and standard deviations of each column"""
    num_rows, num_cols = shape(data_matrix)
    means = [mean(get_column(data_matrix,j))
             for j in range(num_cols)]
    stdevs = [standard_deviation(get_column(data_matrix,j))
              for j in range(num_cols)]
    return means, stdevs
```

然后用结果创建新的数据矩阵：

```
def rescale(data_matrix):
    """rescales the input data so that each column
    has mean 0 and standard deviation 1
    leaves alone columns with no deviation"""
    means, stdevs = scale(data_matrix)

    def rescaled(i, j):
        if stdevs[j] > 0:
            return (data_matrix[i][j] - means[j]) / stdevs[j]
        else:
            return data_matrix[i][j]

    num_rows, num_cols = shape(data_matrix)
    return make_matrix(num_rows, num_cols, rescaled)
```

一如既往地，你需要运用你的判断力。如果你拿到一个由身高和体重组成的巨大的数据集，需要将其过滤为仅由身高在 69.5 英寸至 70.5 英寸之间的人组成。很有可能（取决于你希望回答的问题）剩余的变差仅仅是噪声（noise），但你也许并不希望将其标准差与其他维度的标准差等而视之。

10.5　降维

有时候，数据的“真实”（或有用的）维度与我们掌握的数据维度并不相符。比如，考虑图 10-5 中所示的数据。

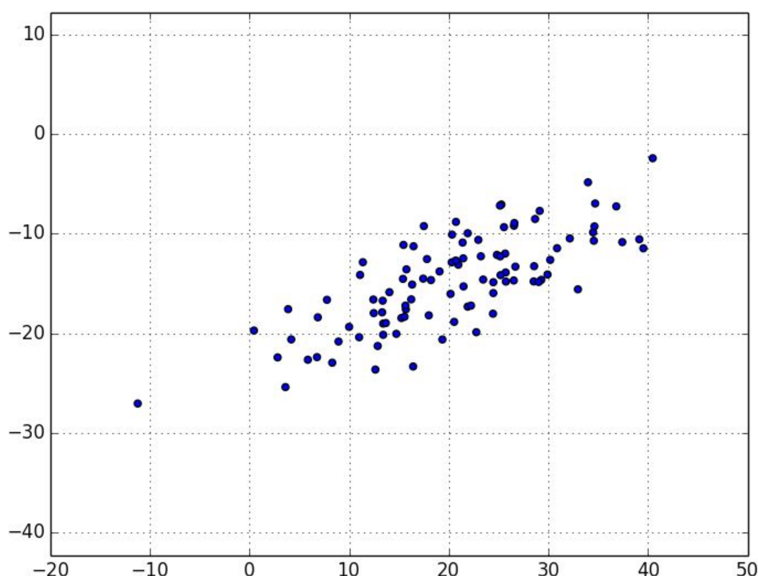


图 **10-5**：坐标轴“错误”的数据

数据的大部分变差看起来像是沿着单个维度分布的，既不与 x 轴对应，也不与 y 轴对应。

当这种情形发生时，我们可以使用一种叫作主成分分析（principal component analysis，PCA）的技术从数据中提取出一个或多个维度，以捕获数据中尽可能多的变差。



实际上，这样的技术不适用于低维数据集。降维

多用于数据集的维数很高的情形，你可以通过一个小子集来抓住数据集本身的大部分变差。不过，这种情况很复杂，绝非一两章能讲得清。

首先，我们需要将数据转换成为每个维度均值为零的形式：

```
def de_mean_matrix(A):
    """returns the result of subtracting from every value in A the mean
    value of its column. the resulting matrix has mean 0 in every column"""
    nr, nc = shape(A)
    column_means, _ = scale(A)
    return make_matrix(nr, nc, lambda i, j: A[i][j] - column_means[j])
```

（如果不这样做，应用这种技术的结果可能就只是确定数据的均值本身，而非找出数据中的变差。）

图 10-6 展示了去均值后的示例数据。

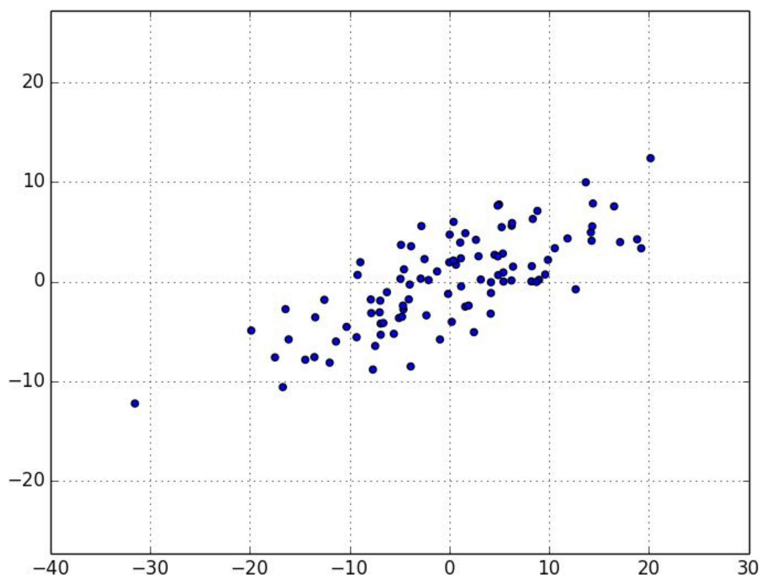


图 **10-6**：去均值后的数据

现在，已有一个去均值的矩阵 X，我们想问，最能抓住数据最大变差的方向是什么？

具体来说，给定一个方向 d（一个绝对值为 1 的向量），矩阵的每行

x 在方向 d 的扩展是点积 `dot(x, d)`。并且如果将每个非零向量 w 的绝对值大小调整为 1，则它们每个都决定了一个方向：

```
def direction(w):
    mag = magnitude(w)
    return [w_i / mag for w_i in w]
```

因此，已知一个非零向量 w，我们可以计算 w 方向上的方差：

```
def directional_variance_i(x_i, w):
    """the variance of the row x_i in the direction determined by w"""
    return dot(x_i, direction(w)) ** 2

def directional_variance(X, w):
    """the variance of the data in the direction determined w"""
    return sum(directional_variance_i(x_i, w)
               for x_i in X)
```

我们可以找出使方差最大的那个方向。只要得到梯度函数，我们就可以通过梯度下降法计算出来：

```
def directional_variance_gradient_i(x_i, w):
    """the contribution of row x_i to the gradient of
    the direction-w variance"""
    projection_length = dot(x_i, direction(w))
    return [2 * projection_length * x_ij for x_ij in x_i]

def directional_variance_gradient(X, w):
    return vector_sum(directional_variance_gradient_i(x_i,w)
                      for x_i in X)
```

第一主成分仅是使函数 `directional_variance` 最大化的方向：

```
def first_principal_component(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_batch(
        partial(directional_variance, X),           # 现在是w的一个函数
        partial(directional_variance_gradient, X),  # 现在是w的一个函数
        guess)
    return direction(unscaled_maximizer)
```

也许，你也有可能使用随机梯度下降方法：

```
# 这里没有"y"，所以我们仅仅是传递一个Nones的向量
# 和忽略这个输入的函数
def first_principal_component_sgd(X):
    guess = [1 for _ in X[0]]
    unscaled_maximizer = maximize_stochastic(
        lambda x, _, w: directional_variance_i(x, w),
        lambda x, _, w: directional_variance_gradient_i(x, w),
        X,
        [None for _ in X],    # 假的 "y"
        guess)
    return direction(unscaled_maximizer)
```

对去均值的数据集，计算结果返回了方向 [0.924, 0.383]，这个方向看起来捕获了数据变动的主要方向轴（图 10-7）。

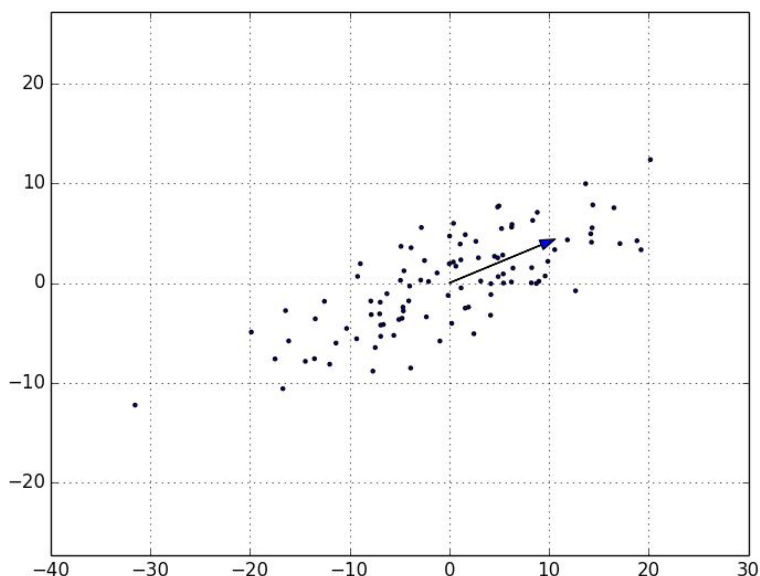


图 **10-7**：第一主成分

一旦我们找到了第一主成分的方向，就可以将数据在这个方向上投影得到这个成分的值：

```
def project(v, w):
    """return the projection of v onto the direction w"""
    projection_length = dot(v, w)
    return scalar_multiply(projection_length, w)
```

如果还想得到其他的成分，就要先从数据中移除投影：

```
def remove_projection_from_vector(v, w):
    """projects v onto w and subtracts the result from v"""
    return vector_subtract(v, project(v, w))

def remove_projection(X, w):
    """for each row of X
    projects the row onto w, and subtracts the result from the row"""
    return [remove_projection_from_vector(x_i, w) for x_i in X]
```

因为这个例子中的数据集仅仅设定为二维，当移除第一主成分之后，剩下的实际上就是一个一维的成分了（图 10-8）。

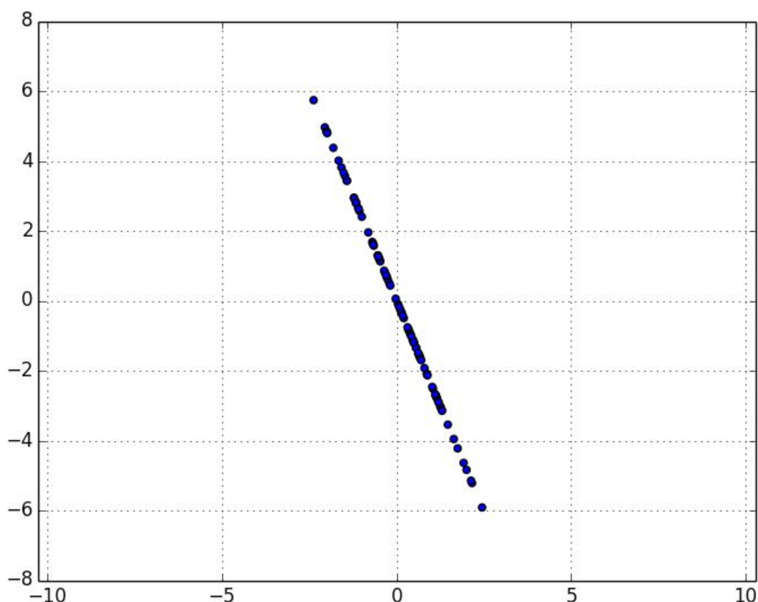


图 **10-8**：移除第一主成分之后的数据

在这点上，我们可以通过对 `remove_projection` 的结果重复这个过程来找到其他的主成分（图 10-9）。

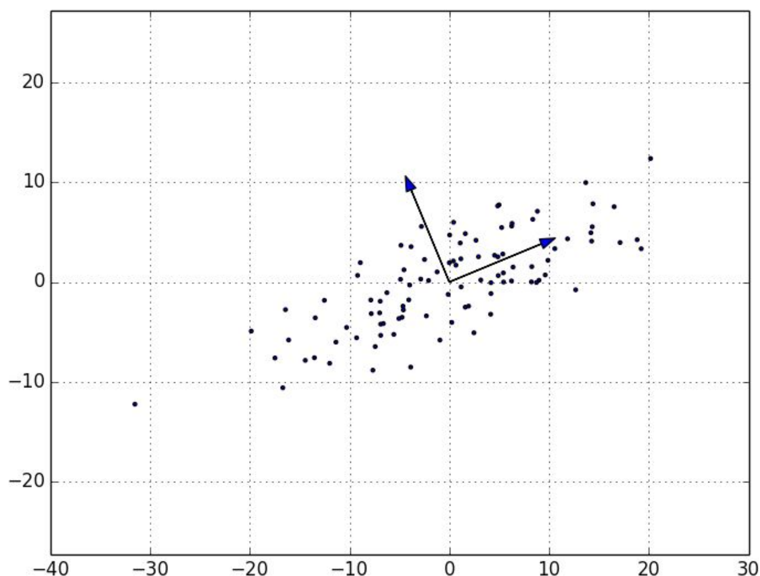


图 **10-9**：前两个主成分

在更高维的数据集中，我们可以通过迭代找到我们所需的任意数目的主成分：

```
def principal_component_analysis(X, num_components):
    components = []
    for _ in range(num_components):
        component = first_principal_component(X)
        components.append(component)
        X = remove_projection(X, component)

    return components
```

然后再将原数据转换为由主成分生成的低维空间中的点：

```
def transform_vector(v, components):
    return [dot(v, w) for w in components]

def transform(X, components):
    return [transform_vector(x_i, components) for x_i in X]
```

这种技术很有价值，原因有以下几点。首先，它可以通过清除噪声维度和整合高度相关的维度来帮助我们清理数据。

第二，在提取出数据的低维代表后，我们就可以运用一系列并不太适用于高维数据的技术。我们可以在本书的很多地方看到运用这种技术的例子。

同时，它既可以帮助你建立更棒的模型，又会使你的模型更难理解。很容易理解诸如“工作年限每增加一年，平均工资会增加 1 万美元”这样的结论。但诸如“第三主成分每增加 0.1，平均工资就会增加 1 万美元”这样的结论就很难理解了。

10.6　延伸学习

- 正如我们在第 9 章末所提到的，pandas（http://pandas.pydata.org/）很可能是 Python 清理、整理、处理和利用数据的主要工具。本章中所有自己动手创建的范例都可以通过 pandas 更简单地完成。*Python for Data Analysis*（O'Reilly）大概是学习 pandas 的最好途径。
- scikit-learn 有多种多样的矩阵分解函数（http://scikit-learn.org/stable/modules/classes.html#module-sklearn.decomposition），包括 PCA。

第 11 章　机器学习

耳提面命诚可贵，求知若渴价更高。

——温斯顿 · 丘吉尔

在很多人眼里，数据科学几乎就是机器学习，而数据科学家每天做的事就是建立、训练和调整机器学习模型（而且，这些人当中有很大一部分并不真正知道机器学习是什么）。事实上，数据科学的主要内容是把商业问题转换为数据问题，然后收集数据、理解数据、清理数据、整理数据格式，而后才轮到机器学习这一后续工作。尽管如此，机器学习也是一种有趣且必要的后续工作，为了做好数据科学工作，你很有必要学习它。

11.1　建模

在讨论机器学习之前，我们需要谈谈模型（model）。

什么是模型？它实际上是针对存在于不同变量之间的数学（或概率）联系的一种规范。

比如，如果你想为你的社交网站融资，可以建立一个商业模型（大多数情况下建立在一个工作表里），模型的输入是诸如“用户数”“每位用户的广告收入”“雇员数”之类的变量，输出是接下来几年的年度利润。某本烹调指南涉及的模型是输入“吃饭的人数”和“饥饿的程度”来量化所需要的材料。如果你在电视上看过扑克比赛，就会知道选手们通过一个记牌的模型来实时地估计每位玩家的“获胜概率”，这个模型考虑了已经出的牌和还在牌桌上的牌的分布。

商业模型很可能是建立在简单的数学联系上：利润是收入减去支出，收入是平均价格乘以单位销售量，诸如此类。菜谱模型则可能基于反复试验之上——某人走进厨房，尝试不同的原料组合，直到发现自己喜欢的口味。扑克模型基于概率论、扑克规则和某些关于处理牌的随机过程的合理假设。

11.2　什么是机器学习

关于什么是机器学习，每个人都有自己确切的定义。在这里，我们使用的定义是创建并使用那些由学习数据而得出的模型。在其他语境中，这也可以叫作预测建模或者数据挖掘，但是我们选择使用机器学习这个术语。一般来说，我们的目标是用已存在的数据来开发可用来对新数据预测多种可能结果的模型，比如：

- 预测一封邮件是否是垃圾邮件
- 预测一笔信用卡交易是否是欺诈行为
- 预测哪种广告最有可能被购物者点击
- 预测哪支橄榄球队会赢得超级杯大赛

下面我们会看到有监督的模型（其中的数据标注有正确答案，可供学习）和无监督的模型（没有标注）。还有一些其他类型的模型，我们不会在本书中进行讨论，如半监督的模型（其中有一部分数据带有标注）和在线的模型（模型需要根据新加入的数据做持续调整）。

现在，甚至在最简单的情况下都有一整套模型来描述我们感兴趣的联系。大多数情况下我们会自己选择参数化的模型族，然后使用数据来学习从某种程度上进行优化了的参数。

例如，我们假设一个人的身高（大致上）是他体重的线性函数，然后用数据来学习这个线性函数。或者，我们也可以假设决策树是一种用来诊断患者疾病的好方法，然后使用数据来学习一颗“最优”的树。本书剩余部分会穿插讲述可供我们学习的不同的模型族。

但在此之前，我们需要更好地理解机器学习的基础。本章剩余部分将探讨一些机器学习的基本概念，随后才会讨论到模型本身。

11.3　过拟合和欠拟合

在机器学习中，一种常见的困境是过拟合（overfitting）——指一个在训练数据上表现良好，但对任何新数据的泛化能力却很差的模型。这可能牵扯到对数据中噪声的学习，也可能涉及学习识别特别的输入，而不是对可以得到期望的输出进行准确预测的任何因素。

另一种有害的情况是欠拟合（underfitting），它产生的模型甚至在训练数据上都没有好的表现，尽管通常这暗示你模型不够好而要继续寻找改进的模型。

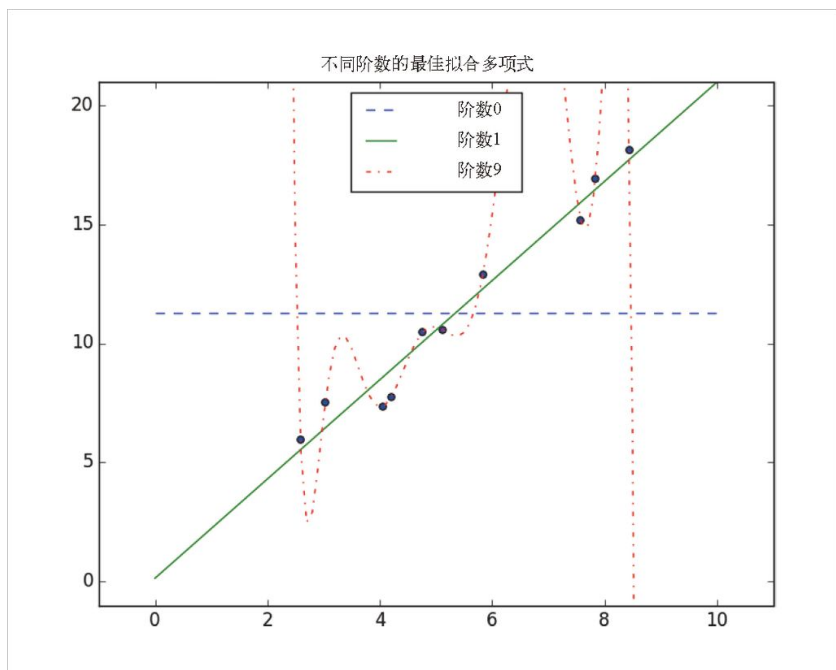


图 **11-1**：过拟合和欠拟合

在图 11-1 中，我对一组简单的数据拟合了 3 种多项式（别担心这是怎么做到的，我们会在后面的章节中介绍）。

水平线显示了最佳拟合阶数为 0（也就是常数）的多项式，它对训练数据来说存在严重的欠拟合。最佳拟合的阶数为 9（也就是有 10 个参数）的多项式精确地穿过训练数据的每个点，但这是严重过拟合的。如果我们能取到更多的一些点，这个多项式很有可能会偏离它们很多。阶数为 1 的线把握了很好的平衡——它和每个点都很接近，并且（如果这些数据是有代表性的）它也会和新的数据点很接近。

很明显，太复杂的模型会导致过拟合，并且在训练数据集之外不能很好地泛化。所以，我们该如何确保我们的模型不会太复杂呢？最基本的方法包括使用不同的数据来训练和测试模型。

最简单的做法是划分数据集，使得（比如说）三分之二的数据用来训练模型，之后用剩余的三分之一来衡量模型的表现：

```
def split_data(data, prob):
    """split data into fractions [prob, 1 - prob]"""
    results = [], []
```

```
    for row in data:
        results[0 if random.random() < prob else 1].append(row)
    return results
```

通常我们会有一个作为输入变量的矩阵 x 和一个作为输出变量的向量 y。这种情况下，我们要确保无论在训练数据还是测试数据中，都要把对应的值放在一起：

```
def train_test_split(x, y, test_pct):
    data = zip(x, y)                                 # 成对的对应值
    train, test = split_data(data, 1 - test_pct)     # 划分这个成对的数据
    x_train, y_train = zip(*train)                   # 魔法般的解压技巧
    x_test, y_test = zip(*test)
    return x_train, x_test, y_train, y_test
```

这样你就可以做一些类似下面这样的处理：

```
model = SomeKindOfModel()
x_train, x_test, y_train, y_test = train_test_split(xs, ys, 0.33)
model.train(x_train, y_train)
performance = model.test(x_test, y_test)
```

如果模型对训练数据是过拟合的，那么它在（完全划分开的）测试数据集上会有可能真的表现得很不好，换句话说，如果它在测试数据上表现良好，那么你可以肯定地说它拟合良好而非过拟合。

然而在有些情况下这也可能会出错。

第一种情况是训练和测试数据集中的共有模式不能泛化到大型数据集上。

比如，假设数据集包括用户活跃度，每位用户每周一列。在此情形下，大多数用户会出现在训练数据和测试数据中，而且有些模型可能会学习识别用户而不是去发现涉及属性的联系。这不是个太大的问题，尽管我曾经遇到过一次。

一个更大的问题是，如果你划分训练集和测试集的目的不仅仅是为了判断模型，也是为了在许多模型中进行选择。此时，尽管不是所有的模型都是过拟合的，但“选择在测试集上表现最好的模型”是一种元训练（meta-training），会把测试集当作另一个训练集而运作。（当然，在测试集上有最好表现的模型会在测试集上有持续的好表现。）

在这种情况下，你应该把数据划分为三部分：一个用来建立模型的训练集，一个为在训练好的模型上进行选择的验证集，一个用来判断最终的模型的测试集。

11.4　正确性

当我不做数据科学的时候，我会涉猎医疗研究。在业余时间，我做了一个低成本、无害的新生儿测试——准确性高达 98%——测试新生儿是否会得白血病。我的律师确信我是有专利权的。所以我在这里把细节分享一下：仅仅当婴儿起名为 Luke 时预测会得白血病（因为这个名字听起来像白血病的英文 leukemia）。

如我们下面所见，这种测试确实有超过 98% 的准确性。然而，这是一个极为愚蠢的测试，也很好地解释了我们为什么不能仅用“准确率”这个概念来测量一个模型的好坏。

假设建立一个模型来做二元的判断，比如：一封邮件是否是垃圾邮件？我们是否该聘用这位应聘者？这个乘客是潜藏的恐怖分子吗？

给定一个标签数据集和一个预测模型，每个数据集都会落在下面其中一个属性中。

- 真阳性：“这封邮件是垃圾邮件，我们做了正确的预测。”
- 假阳性（又称第 1 类错误）：“这封邮件不是垃圾邮件，但是我们预测它是垃圾邮件。”
- 假阴性（又称第 2 类错误）：“这封邮件是垃圾邮件，但是我们预测它不是垃圾邮件。”
- 真阴性：“这封邮件不是垃圾邮件，而且我们正确地预测了它不是垃圾邮件。”

我们常用混淆矩阵（confusion matrix）中的计数来表示上面的四种情况：

	垃圾邮件 / 非垃圾邮件	
预测“是垃圾邮件”		
预测“非垃圾邮件”		

让我们来看看我的白血病测试是如何符合这种框架的。现如今，大约每 1000 名婴儿中有 5 人会起名叫 Luke（http://www.babycenter.com/babyNameAllPops.htm?babyNameId=2918）。每人一生中罹患白血病的概率大约是 1.4%，或者说每 1000 人中会有 14 人患病（http://seer.cancer.gov/statfacts/html/leuks.html）。

如果我们相信这两个因素是独立的，然后对 1 百万人运用我的“Luke 是白血病患者”测试，预计能看到这样的混淆矩阵：

	[illegible]		
[illegible]			
[illegible]			
[illegible]			

然后我们由此计算关于模型表现的多个统计量。例如，准确率（accuracy）定义为正确预测的比例：

```
def accuracy(tp, fp, fn, tn):
    correct = tp + tn
    total = tp + fp + fn + tn
    return correct / total

print accuracy(70, 4930, 13930, 981070)     # 0.98114
```

看起来这个数字令人印象十分深刻，但很明显这并不是一个好的测试，这意味着我们不能对原始的准确率有过多的信心。

更常见的做法是把查准率（precision）和查全率（recall）结合起来看待。查准率度量我的模型所做的关于“阳性”的预测有多准确：

```
def precision(tp, fp, fn, tn):
    return tp / (tp + fp)

print precision(70, 4930, 13930, 981070)     # 0.014
```

查全率度量我的模型所识别的“阳性”的比例：

```
def recall(tp, fp, fn, tn):
    return tp / (tp + fn)
```

```
print recall(70, 4930, 13930, 981070)      # 0.005
```

这两个结果都低得可怕，反映出这是一个很不好的模型。

有时候可以把查准率和查全率组合成 F1 得分（F1 score），它是这样定义的：

```
def f1_score(tp, fp, fn, tn):
    p = precision(tp, fp, fn, tn)
    r = recall(tp, fp, fn, tn)

    return 2 * p * r / (p + r)
```

它是查准率和查全率的调和平均值（https://en.wikipedia.org/wiki/Harmonic_mean），因此必然会落在两者之间。

模型的选择通常是查准率和查全率之间的权衡。一个模型如果在信心不足的情况下预测“是”，那么它的查全率可能会较高，但查准率却较低；而如果一个模型在信心十足的情况下预测“是”，那么它的查全率可能会较低，但查准率却较高。

另一方面，也可以把这当作假阳性和假阴性之间的权衡。预测的“是”太多通常会给出很多的假阳性。预测的“否”太多通常会给出很多的假阴性。

假设对白血病来说有 10 个风险因素，你的身体具备的因素越多，就越容易患上白血病。这种情况下，你可以假设进行一系列连续性的测试：“至少有一个风险因素预测会得白血病”“至少有两个风险因素预测会得白血病”诸如此类。随着临界值的不断提高，测试的查准率也提高了（因为具有更多风险因素的人更容易患上白血病），并且降低了测试的查全率（因为能够达到临界值的最终患病者越来越少）。对于类似这样的情况，选择合适的临界值实际上就是做出正确的权衡。

11.5　偏倚-方差权衡

思考过拟合问题的另一种角度是把它作为偏倚和方差之间的权衡。偏倚和方差这两个名词是用来度量在（来自同一个大型总体的）不同的训练数据集上多次重复训练模型的情况。

比如，在 11.3 节“过拟合和欠拟合”中提到的 0 阶模型对（取自同一

总体的）任何可能的训练集都会造成大量的错误。这表明该模型偏倚较高。然而任何两个随机选择的训练集会给出很相似的模型（因为任何两个随机选择的训练集都应该有大致相似的平均值）。所以我们称这个模型有低方差。高偏倚和低方差典型地对应着欠拟合。

另一方面，9 阶模型完美地拟合训练集，它具有很低的偏倚和很高的方差（因为任何两个训练集都可能给出非常不同的模型形式）。这种情况对应过拟合。

如果你的模型有高偏倚（这意味着即使在训练数据上也表现不好），可以尝试加入更多的特征。从 0 阶模型到 1 阶模型就是一个很大的改进。

如果你的模型有高方差，那可以类似地移除特征；另一种解决方法是（如果可能的话）获得更多的数据。

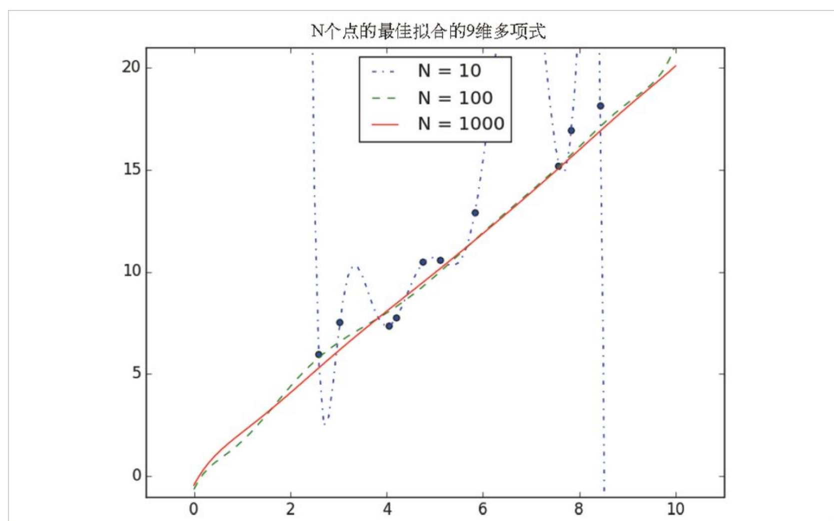


图 **11-2**：利用更多数据降低方差

在图 11-2 中，我们对不同大小的样本拟合了 9 阶多项式。如我们前面所见，基于 10 个点拟合的模型是一塌糊涂的。如果在 100 个数据点上训练，就会大大减少过拟合的问题。如果在 1000 个点上训练，看起来就像是 1 阶模型。

在模型复杂度不变的前提下，你有越多的数据，就越难过拟合。

另一方面，更多的数据对偏倚并不会有帮助。如果模型不能使用足够

多的特征来捕捉数据的正则性，那么再多的数据也不会有帮助。

11.6　特征提取和选择

我们之前提到，如果数据没有足够的特征，模型很可能就会欠拟合。但如果数据有太多的特征，模型又容易过拟合。那什么是特征呢，它们又从何而来？

特征（feature）是指提供给模型的任何输入。在最简单的情况下，特征是直接提供给你的。如果你想基于某人的工作年限来预测其薪水，那工作年限就是你所拥有的唯一的特征。

（尽管如此，如同我们在 11.3 节“过拟合和欠拟合”中所见的，如果可以帮助你建立更好的模型，应该加入工作年限的平方项和立方项。）

当数据变得更复杂时，事情变得有趣起来。设想我们尝试建立一个垃圾邮件过滤器来预测一封邮件是否是垃圾邮件。大多数模型不知道如何处理原始邮件，邮件就是一组文本。你需要提取特征，比如：

- 邮件中是否包含单词“Viagra”；
- 字母 d 出现了多少次；
- 寄件人的域名是什么。

第一个问题的特征就是简单的是或否，可以被典型地编码为 1 或 0。第二个问题的特征是个数字。第三个问题的特征是从一个离散的选项集中做出的选择。

多数情况下，我们会从符合这三种特征的数据中提取特征。此外，特征的类型限制了我们所用模型的类型。

第 13 章中使用的朴素贝叶斯分类器适合“是或否”这样的二元特征，就像上面列出的第一种情况一样。

第 14 章和第 16 章将要提到的回归模型要求有数值型的特征（它可能会包括 0 或 1 这样的虚拟变量）。

第 17 章讲到的决策树，会涉及数值或属性数据。

尽管在垃圾邮件过滤器的例子中我们探索了创建特征的方法，但有时我们还需要设法移除特征。

比如，输入可能是包含几百个数的向量。根据具体情况，最好是取出一些维度，缩减到只剩少量重要的维度（见 10.5 节“降维”），只使用这些少数的特征，或者最好采用一些技术（比如正则化技术，见 15.8 节“正则化”）对应用过多特征的模型进行惩罚。

我们该如何选择特征呢？这需要经验和专业知识的结合。如果你收到了大量的邮件，可能会对某个特定的词比较敏感，这个词会成为垃圾邮件的好指标。同时，你也可能会觉得，字母 d 的个数不像是判断垃圾邮件的好指标。但通常来说，你需要尝试不同的特征，这也不失为一种乐趣。

11.7　延伸学习

- 继续读下去，后面几章讨论了不同种类的机器学习模型。
- Coursera 的机器学习课程（https://www.coursera.org/learn/machine-learning）是原创的 MOOC，是深入理解机器学习基础知识的好途径。加州理工学院的机器学习 MOOC（https://work.caltech.edu/telecourse.html）也是很好的资源。
- *The Elements of Statistical Learning* 是一本相当权威的教材，可以从网络上免费下载（http://statweb.stanford.edu/~tibs/ElemStatLearn/）。但是要注意，它是非常数学化的。

第 12 章　*k* 近邻法

惹怒你的邻居很容易，只需坦言你对他们的真实印象。

——皮特罗 · 雷提诺

假设你打算预测我会在下次大选给谁投票。如果你对我一无所知（但是你有数据），一个明智的方法是看看我的邻居们打算怎么投票。我住在西雅图市中心，我的邻居们总是会打算投票给民主党的候选人，这说明“民主党候选人”有可能是我的投票对象。

现在假设你对我的了解不仅限于家庭住址——可能你还知道我的年龄、收入，以及有几个孩子，等等。参照我的行为被这些维度影响（或刻画）的程度，观察那些在这些维度上最接近我的邻居似乎比观察我所有的邻居会得到更好的预测结果。这就是最近邻分类（nearset neighbors classification）方法背后的思想。

12.1　模型

最近邻法是最简单的预测模型之一，它没有多少数学上的假设，也不要求任何复杂的处理，它所要求的仅仅是：

- 某种距离的概念
- 一种彼此接近的点具有相似性质的假设

本书中讲到的大部分技术是把数据集看作一个整体，以便学习数据中的模式。相比之下，最近邻法却非常有意地忽略了大量信息，因为对每一个新的数据点进行预测只依赖于少量最接近它的点。

而且，最近邻法并不能帮助理解你所观察到的任意现象的驱动机制。比如，基于我邻居的投票行为来预测我的投票并不能告诉你我为什么要这样投票，而某些基于（比如说）我的收入和婚姻状况来预测我投票行为的模型很可能会揭示我投票的原因。

在一般情形中，我们有一些数据和对应的标签集。这些标签可能记作真或假，表示每个输入是否满足诸如“是否是垃圾邮件”“是否有毒”“是否值得观看”这样的条件；或者它们可能表示属性，就像“G、PG、

PG-13、R、PG-17”这样的电影评分；或者它们可能是总统候选人的名字；或者它们可能是最受欢迎的编程语言的名称。

在我们的例子里，数据点是向量，这意味着可以用到第 4 章中的 distance 函数。

比如说我们已选定了数字 k 的值为 3 或 5，然后想要对某些新的数据点分类时，我们寻找 k 个已标记的最接近它的点，让这些点在新的输出上投票。

为此，我们需要一个函数来计算投票结果。一个可能的函数是：

```
def raw_majority_vote(labels):
    votes = Counter(labels)
    winner, _ = votes.most_common(1)[0]
    return winner
```

但这里没有智能地处理并列的结果。比如，假设我们在对电影评分，且 5 个最接近的电影被评为 G、G、PG、PG 和 R，那么 G 有两票，PG 也有两票。在这种情形下，我们有以下几种选择。

- 随机选择其中一个获胜者。
- 根据距离加权投票并选择加权的获胜者。
- 减少 k 值直到找到唯一的获胜者。

我们将运用第三种方法：

```
def majority_vote(labels):
    """assumes that labels are ordered from nearest to farthest"""
    vote_counts = Counter(labels)
    winner, winner_count = vote_counts.most_common(1)[0]
    num_winners = len([count
                       for count in vote_counts.values()
                       if count == winner_count])

    if num_winners == 1:
        return winner                     # 唯一的获胜者，返回它的值
    else:
        return majority_vote(labels[:-1]) # 去掉最远元素，再次尝试
```

这种方法最终是管用的，因为在最坏的情况下，我们会一直减少 k

值，直到只剩一个标签，此时这个标签就是获胜者。

使用这个函数很容易创建一个分类器：

```
def knn_classify(k, labeled_points, new_point):
    """each labeled point should be a pair (point, label)"""

    # 把标记好的点按从最近到最远的顺序排序
    by_distance = sorted(labeled_points,
                         key=lambda (point, _): distance(point, new_point))

    # 寻找k个最近邻的标签
    k_nearest_labels = [label for _, label in by_distance[:k]]

    # 然后让它们投票
    return majority_vote(k_nearest_labels)
```

让我们看看这是怎么起作用的。

12.2　案例：最喜欢的编程语言

DataSciencester 第一次用户调查的结果回来了，我们从中找到了一系列大城市用户偏爱的编程语言：

```
# 每一条记录都是([longitude, latitude], favorite_language)的形式

cities = [([-122.3 , 47.53], "Python"),  # 西雅图
          ([-96.85, 32.85], "Java"),     # 奥斯汀
          ([ -89.33, 43.13], "R"),       # 麦迪逊
          # ……还有很多记录
]
```

社区参与部门的副总想知道我们能不能用这些结果来预测那些我们没有调查到的地方最喜欢的编程语言是什么。

一如既往地，第一步最好是先根据数据作图（如图 12-1 所示）：

```
# 键是语言，值是成对数据(longitudes, latitudes)
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

# 我们希望每种语言都能有不同的记号和颜色
markers = { "Java" : "o", "Python" : "s", "R" : "^" }
colors  = { "Java" : "r", "Python" : "b", "R" : "g" }
```

```
for (longitude, latitude), language in cities:
    plots[language][0].append(longitude)
    plots[language][1].append(latitude)

# 对每种语言创建一个散点序列
for language, (x, y) in plots.iteritems():
    plt.scatter(x, y, color=colors[language], marker=markers[language],
                      label=language, zorder=10)

plot_state_borders(plt)         # 假设我们有一个实现这一步的函数

plt.legend(loc=0)               # 让matplotlib选择一个位置
plt.axis([-130,-60,20,55])      # 设置轴

plt.title("最受欢迎的编程语言")
plt.show()
```

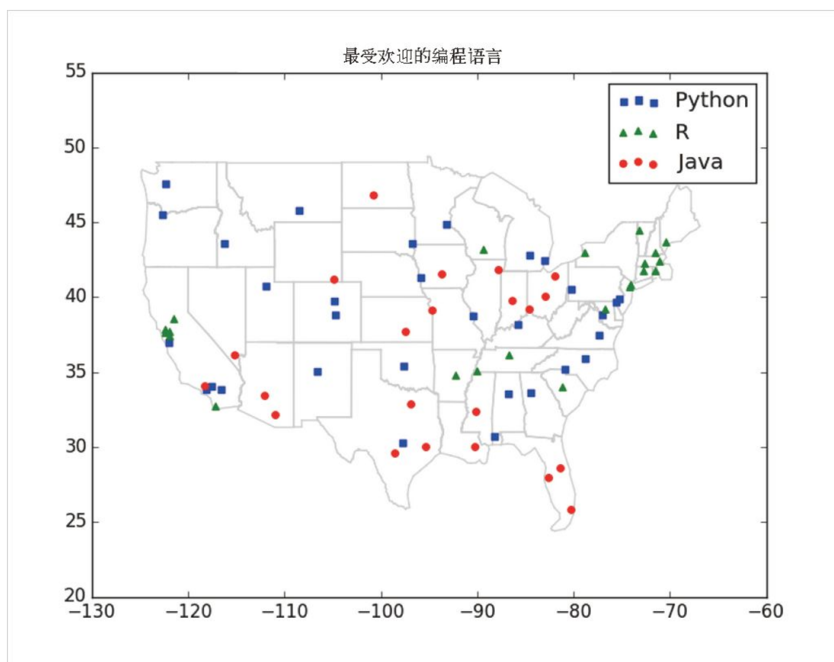


图 12-1：最受欢迎的编程语言



你可能注意到了对 `plot_state_borders()` 的调用，这是一个没有被精确定义的函数。本书的 GitHub 页面（https://github.com/joelgrus/data-

science-from-scratch）上有这个函数的具体实现，你也可以把它当作一个练习题来自己尝试解决：

(1) 在网络上搜索各州边界线的经纬度等信息；

(2) 把你找到的任意经纬度数据转化为线段 [(long1, lat1), (long2, lat2)] 的列表；

(3) 使用 `plt.plot()` 画出这些线段。

既然互相邻近的地区看起来偏爱相同的语言，那么 *k* 近邻法作为一种预测模型看上去会是一种合理的选择。

首先，让我们看一下如果尝试利用邻居城市来预测每个城市偏爱的语言会得到什么结果：

```
# 试试多个不同的k值
for k in [1, 3, 5, 7]:
    num_correct = 0

    for city in cities:
        location, actual_language = city
        other_cities = [other_city
                        for other_city in cities
                        if other_city != city]

        predicted_language = knn_classify(k, other_cities, location)

        if predicted_language == actual_language:
            num_correct += 1

    print k, "neighbor[s]:", num_correct, "correct out of", len(cities)
```

看起来 3- 近邻的表现最好，59% 的时间都能给出正确结果：

```
1 neighbor[s]: 40 correct out of 75
3 neighbor[s]: 44 correct out of 75
5 neighbor[s]: 41 correct out of 75
7 neighbor[s]: 35 correct out of 75
```

现在可以看出在每个最近邻体系下会把某个区域分类到哪种语言。我们可以在全部的网格点上进行这种分类，然后参照处理城市分类的方法把预测结果画出来：

```
plots = { "Java" : ([], []), "Python" : ([], []), "R" : ([], []) }

k = 1 # 或3，或5，或……

for longitude in range(-130, -60):
    for latitude in range(20, 55):
        predicted_language = knn_classify(k, cities, [longitude, latitude])
        plots[predicted_language][0].append(longitude)
        plots[predicted_language][1].append(latitude)
```

例如，图 12-2 显示了当我们只看最近的邻居（$k=1$）时会有什么结果。

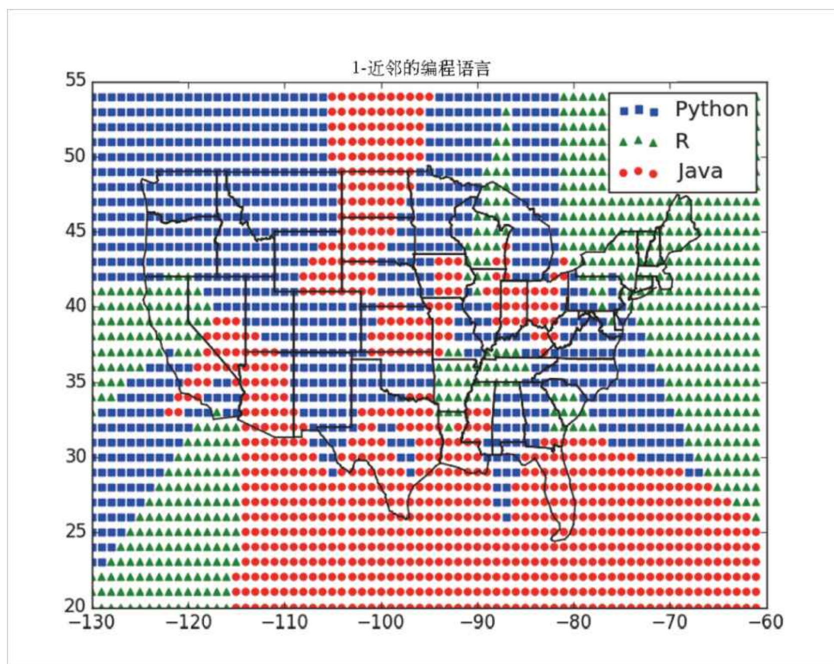


图 **12-2**：**1-** 近邻的编程语言

可以看到，从一种语言到另一种语言有许多骤变，它们之间的边界也较为锐化。当我们把邻居数增加到 3 时，能看到各种语言的区域变光滑了（图 12-3）。

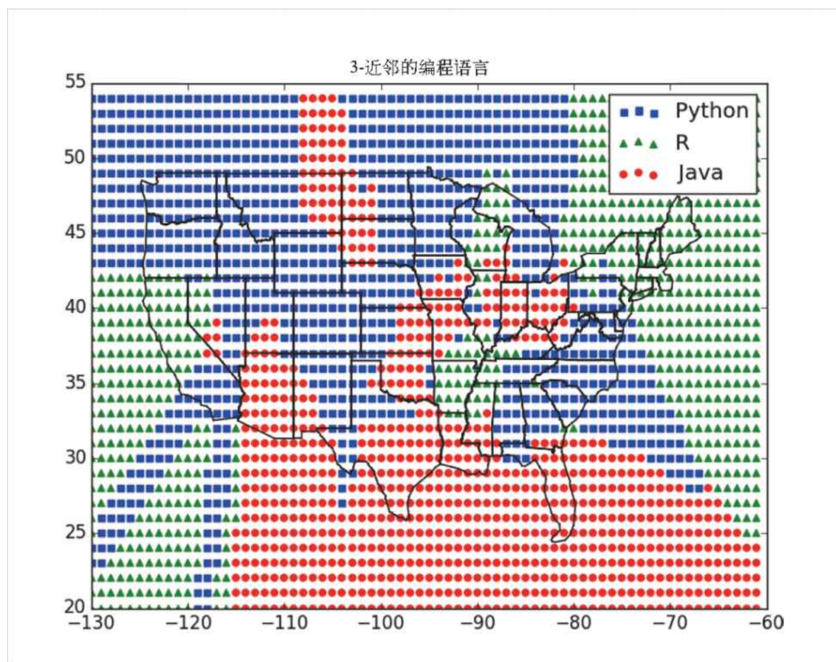


图 **12-3**：**3-** 近邻的编程语言

我们把邻居数增加到 5，边界变得更加光滑了（图 12-4）。

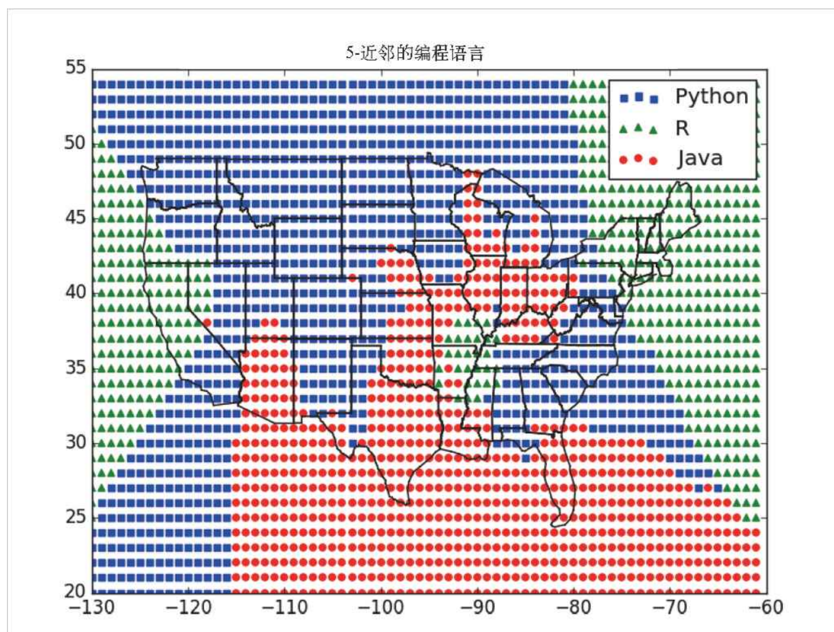


图 **12-4**：**5-** 近邻的编程语言

在这里，维度是大致可比较的，如果不是这样，你可能需要重新调整数据，就像我们在 10.4 节“数据调整”中所讲的那样。

12.3　维数灾难

在更高的维度上，*k* 近邻法会因为“维数灾难”而遇到麻烦，其根源在于高维空间过于巨大。高维空间内的点根本不会表现得彼此邻近。观察维数灾难的一种方法是在一个高维度的 d 维空间“单位立方体”上随机地生成数据点对，并计算它们之间的距离。

现在，生成随机点应该是老生常谈了：

```
def random_point(dim):
    return [random.random() for _ in range(dim)]
```

写一个函数生成距离也是如此：

```
def random_distances(dim, num_pairs):
    return [distance(random_point(dim), random_point(dim))
```

```
          for _ in range(num_pairs)]
```

对从 1 到 100 的每一个维度，我们会计算 10 000 个距离，并使用它们计算每个维度上点和点之间的平均距离和最小距离（图 12-5）：

```
dimensions = range(1, 101)

avg_distances = []
min_distances = []

random.seed(0)
for dim in dimensions:
  distances = random_distances(dim, 10000)    # 10 000个随机对
  avg_distances.append(mean(distances))       # 追踪平均值
  min_distances.append(min(distances))        # 追踪最小值
```

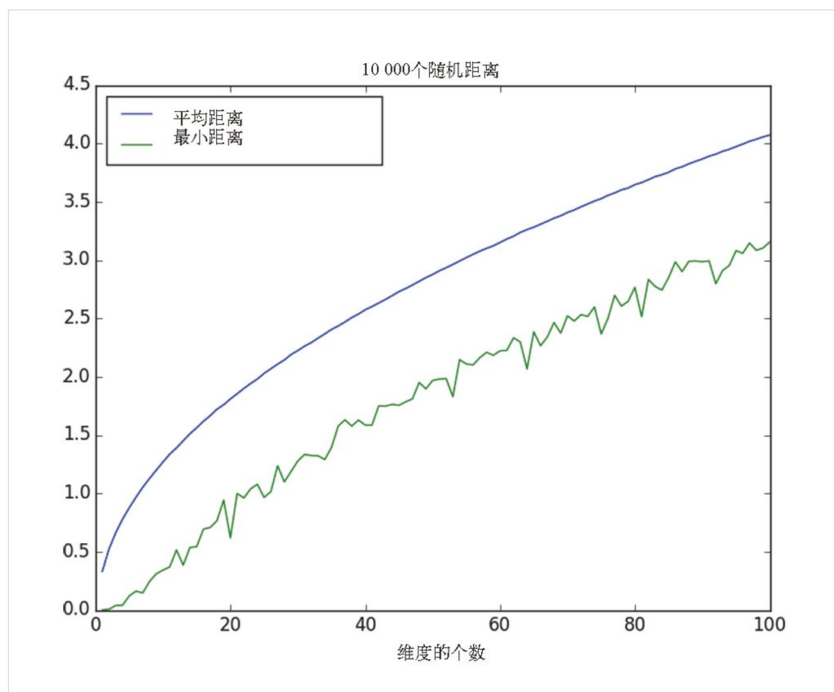


图 **12-5**：维数的灾难

随着维度数量的增加，点和点之间的平均距离也增加了。但更麻烦的是最近距离和平均距离之间的比例（图 12-6）：

```
min_avg_ratio = [min_dist / avg_dist
                 for min_dist, avg_dist in zip(min_distances, avg_distance
```

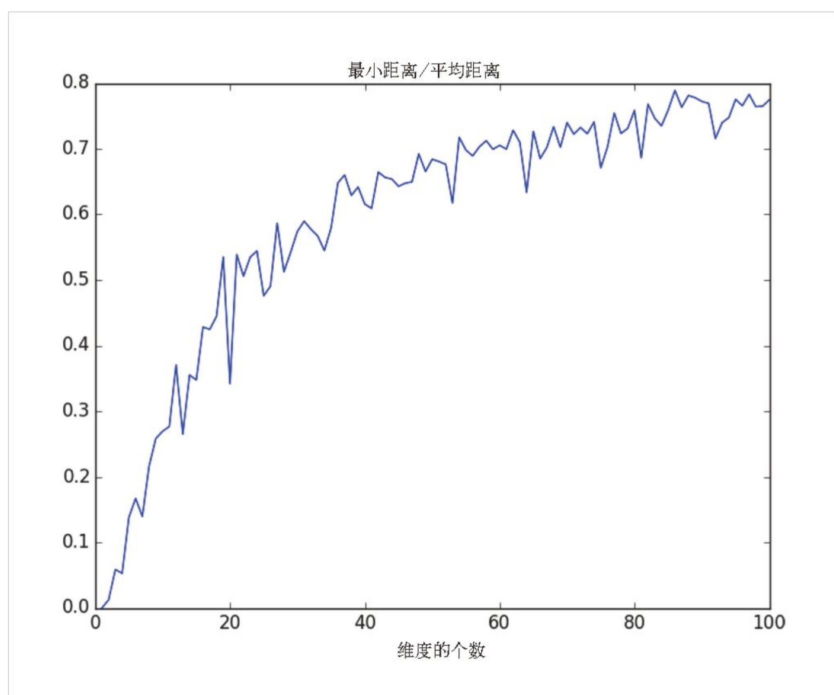


图 **12-6**：维数的另一个灾难

在低维数据集中，最邻近的点的距离看起来比点和点的平均距离要更小。但仅当两个点在每个维度上都邻近时，我们才可称这两个点是邻近的，而且每个增加的维度——即使仅仅是噪声——都有可能会让每个点更加远离其他的点。当有许多维度时，看上去最邻近的两个点的距离并不比点和点的平均距离小，这说明两个点邻近并不特别意味着什么（数据中有许多结构的行为使其看起来像是在更低的维度）。

思考这个问题的一个不同的方法涉及更高维空间的稀疏性。

如果从 0 到 1 之间随机取 50 个数，你可能会得到单位区间内的一个非常好的样本（图 12-7）。

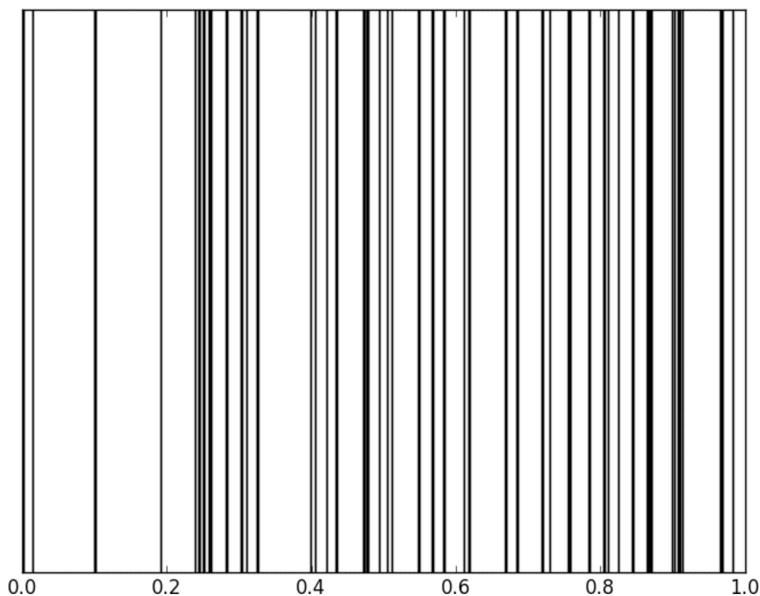


图 **12-7**：一维内的 **50** 个随机点

如果在单位正方形内随机取 50 个点，得到的规模会更小（图 12-8）。

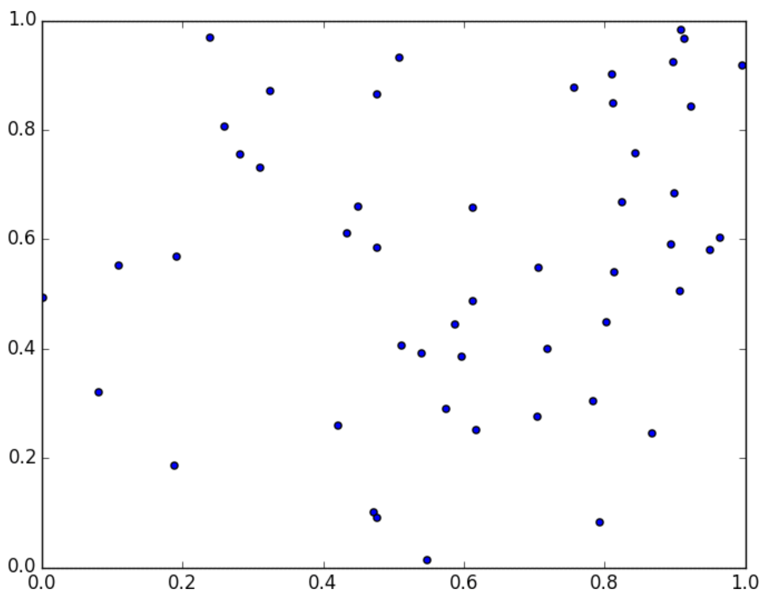


图 **12-8**：二维内的 **50** 个随机点

在三个维度中的随机样本会变得更稀疏（图 12-9）。

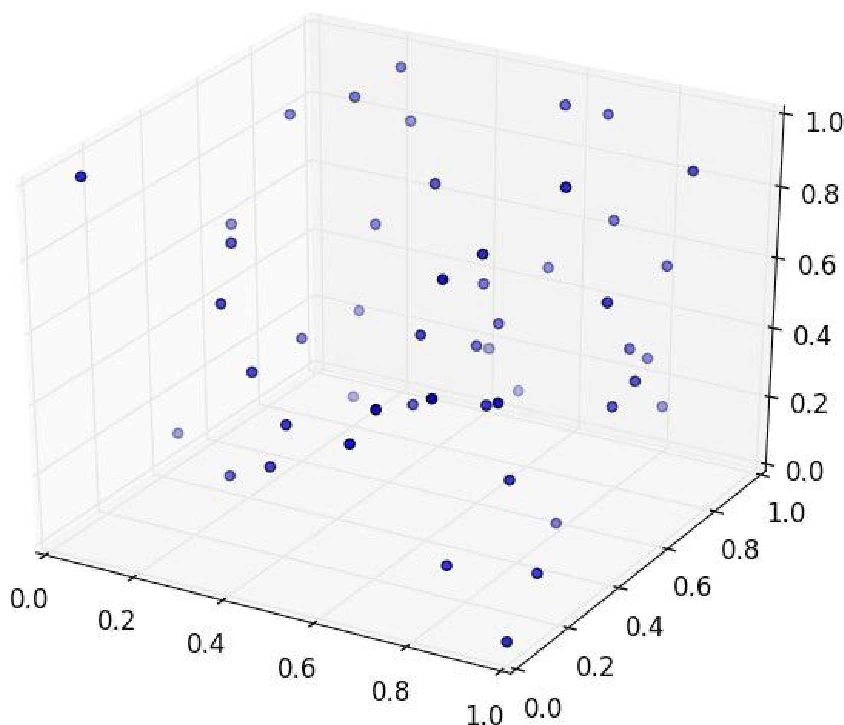


图 12-9：三维内的 **50** 个随机点

`matplotlib` 不能很好地画出 4 维的图形，所以我们只能给出上面的几种情形，即便如此，你也已经能够看到某些点的附近因为没有邻近的点而存在大片的空白空间。在更高的维度上——除非你能以指数规模得到更多的数——大片空白空间代表的是远离你想用在预测中的所有的点的区域。

因此，如果你打算在高维中使用最近邻法，不妨先做一些降维工作。

12.4　延伸学习

scikit-learn 里有许多最近邻模型（http://scikit-learn.org/stable/modules/neighbors.html）。

第 13 章　朴素贝叶斯算法

心灵朴素多为巧，头脑朴素却成拙。

——阿纳托尔 · 法郎士

如果人们不形成关系网的话，社交网络就不会有太大的用途。因此，DataSciencester 提供了一个非常流行的功能，允许会员之间相互发送邮件。虽然大部分会员都是发邮件嘘寒问暖的良民，但是，总少不了有几个坏家伙，老给其他会员发送垃圾邮件，如致富经、药品广告以及以收费为目的的数据科学家资格认证项目等。因此，用户们开始投诉。不久，邮件服务部的副总就把你叫过去，让你利用数据科学来过滤这些垃圾邮件。

13.1　一个简易的垃圾邮件过滤器

想象有一个“全集”，其中存放了从所有可能的邮件中随机选择的邮件。我们令 S 表示事件“这是一封垃圾邮件”，令 V 表示事件“该邮件含有单词 viagra”。在已知邮件中含有单词 viagra 的情况下，该邮件是垃圾邮件的概率可以通过贝叶斯定理求出：

$$P(S|V) = [P(V|S)P(S)]/[P(V|S)P(S) + P(V|\neg S)P(\neg S)]$$

上式中，分子表示某邮件为垃圾邮件并且其中包含单词 viagra 的概率，而分母表示邮件中出现单词 viagra 的概率。因此，你可以认为，上面的公式实际上是在计算兜售伟哥的垃圾邮件所占的比例。

如果我们已经收集了大量垃圾邮件和非垃圾邮件，那么就可以轻松计算 $P(V \mid S)$ 和 $P(V|\neg S)$。如果我们进一步假定任何邮件是垃圾邮件或非垃圾邮件的可能性是等同的（即 $P(S) = P(\neg S) = 0.5$），那么：

$$P(S|V) = P(V|S)/[P(V|S) + P(V|\neg S)]$$

举例来说，如果 50% 的垃圾邮件都含有单词 viagra，而只有 1% 的非垃圾邮件含有该单词，那么任何一封含有单词 viagra 的电子邮件为垃圾邮件的概率是：

$$0.5/(0.5 + 0.01) = 98\%$$

13.2 一个复杂的垃圾邮件过滤器

假设我们已建立了一个词汇表，其中含有许多单词：$w_1, \cdots, w_n$。站在概率论的角度，我 们用 X_i 表示事件“一封含有单词 w_i 的邮件”。此外，我们还假设已经求出了 $P(X_i|S)$ 和 $P(X_i|\neg S)$，前者表示垃圾邮件中出现第 i 个单词的概率，后者表示非垃圾邮件中出现第 i 个单词的概率。

朴素贝叶斯算法的一个（大的）假设是，给定邮件是或不是垃圾邮件的条件下，其中的每个单词存在与否与其他单词毫不相干。直观地讲，就是知道某封垃圾邮件是否含有单词 viagra 无法帮助我们判断该垃圾邮件是否含有单词 rolex。如果用数学公式表示的话，就是：

$$P(X_1 = x_1, \cdots, X_n = x_n|S) = P(X_1 = x_1|S) \times \cdots \times P(X_n = x_n|S)$$

这是一个非常极端的假设（这也部分解释了为何该算法名中含有“朴素”一词）。假设我们的词汇表仅含有单词 viagra 和 rolex，并且一半的垃圾邮件是推销“廉价伟哥”的，另一半是推销“劳力士正品”的，这样的话，我们可以通过朴素贝叶斯算法计算垃圾邮件中同时出现 viagra 和 rolex 这两个单词的概率：

$$P(X_1 = 1, X_2 = 1|S) = P(X_1 = 1|S)P(X_2 = 1|S) = 0.5 \times 0.5 = 0.25$$

之所以得到这样的结果，是因为我们的假设已经把 viagra 和 rolex 绝不会同时出现的经验给扔掉了。尽管这个假设与事实并不相符，但是这个模型的表现通常都很好，所以现实中经常用它过滤垃圾邮件。

前面我们曾经利用贝叶斯定理过滤只涉及 viagra 的垃圾邮件，下面我们再次用它来推断一个邮件是垃圾邮件的概率，具体公式如下所示：

$$P(S|X = x) = P(X = x|S)/[P(X = x|S) + P(X = x|\neg S)]$$

朴素贝叶斯假设使我们能够轻松求出公式右边的每个概率：只要将词汇表中各个单词的概率相乘即可。

在实践中，为了避免所谓的下溢（underflow）问题，你通常希望尽量避免出现大量概率相乘的情况，因为计算机不擅长处理非常接近于零的浮点数。根据代数知识我们知道，$\log(ab) = \log a + \log b$ 且 $\exp(\log x) = x$，因此我们一般使用对浮点数更加友好的等效方法

来计算 $p_1^* \cdots ^* p_n$，具体公式如下所示：

$$\exp(\log(p_1) + \cdots + \log(p_n))$$

现在，唯一的挑战就是估计 $P(X_i|S)$ 和 $P(X_i|\neg S)$ 了，即估计垃圾邮件（或非垃圾邮件）中包含单词 w_i 的概率。如果我们掌握了相当数量的“训练”邮件，即标记为垃圾或非垃圾的邮件，那么很明显，这时计算 $P(X_i|S)$ 就简化为求包含单词 w_i 的垃圾邮件所占的比例了。

但是，这会引起一个大麻烦。假如词汇表中的单词 data 仅出现在训练集的非垃圾邮件中，那么 $P(\text{“data”}|S) = 0$。也就是说，对于任何含有单词 data 的邮件，我们的朴素贝叶斯分类器总是认为它是垃圾邮件的概率为 0，即使是像含有“data on cheap viagra and authentic rolex watches”（关于廉价伟哥和劳力士正品的数据）这样的邮件也是如此。为了避免这种问题，我们通常要使用某种平滑技术。

准确地说，我们引入一个伪记数（pseudo count），记为 k，并通过下面的公式来计算在一封垃圾邮件中出现第 i 个单词的概率：

$P(X_i|S)$ =（k + 含有 w_i 的垃圾邮件的数量）/（$2k$ + 垃圾邮件数量）

$P(X_i|\neg S)$ 的计算方法与此类似。亦即，当计算第 i 个单词出现在垃圾邮件中的概率时，我们假定还看到：额外 k 封垃圾邮件包含该单词，额外 k 封垃圾邮件不包含该单词。

例如，如果 data 这个单词在 98 封垃圾邮件中出现了 0 次，并且 k 取值为 1，我们算出 $P(\text{“data”}|S)$ 为 $1/100 = 0.01$，这样一来，我们的分类器就能给那些含有单词 data 的邮件为垃圾邮件的概率赋予非 0 值了。

13.3　算法的实现

到目前为止，我们已经学习了构建垃圾邮件分类器所需的各方面的知识。下面，我们首先建立一个简单的函数，来将邮件解析为不同的单词。首先要把各个邮件文本转换为小写形式，然后使用 `re.findall()` 提取由字母、数字和撇号组成的“单词”，最后使用 `set()` 函数获得不同的单词：

```
def tokenize(message):
    message = message.lower()                          # 转换为小写
    all_words = re.findall("[a-z0-9']+", message)      # 提取单词
    return set(all_words)                              # 移除副本
```

我们的第二个函数用来计算单词出现在已做标记的邮件训练集中的次数。该函数将返回一个字典，其键为单词，其值为列表，该列表包含两个元素 [spam_count, non_spam_count]，分别表示该单词出现在垃圾邮件和非垃圾邮件中的次数。

```
def count_words(training_set):
    """training set consists of pairs (message, is_spam)"""
    counts = defaultdict(lambda: [0, 0])
    for message, is_spam in training_set:
        for word in tokenize(message):
            counts[word][0 if is_spam else 1] += 1
    return counts
```

接下来，我们利用前面讲过的平滑技术将这些计数转换为估计概率。函数将返回一个列表，列表元素包含三方面的内容，分别是各个单词、该单词出现在垃圾邮件中的概率以及该单词出现在非垃圾邮件中的概率：

```
def word_probabilities(counts, total_spams, total_non_spams, k=0.5):
    """turn the word_counts into a list of triplets
    w, p(w | spam) and p(w | ~spam)"""
    return [(w,
             (spam + k) / (total_spams + 2 * k),
             (non_spam + k) / (total_non_spams + 2 * k))
             for w, (spam, non_spam) in counts.iteritems()]
```

最后要做的事情是利用这些单词的概率（以及朴素贝叶斯假设）给邮件赋予概率：

```
def spam_probability(word_probs, message):
    message_words = tokenize(message)
    log_prob_if_spam = log_prob_if_not_spam = 0.0

    # 迭代词汇表中的每一个单词
    for word, prob_if_spam, prob_if_not_spam in word_probs:

        # 如果*word*出现在了邮件中
        # 则增加看到它的对数概率
        if word in message_words:
```

```
            log_prob_if_spam += math.log(prob_if_spam)
            log_prob_if_not_spam += math.log(prob_if_not_spam)

        # 如果*word*没有出现在邮件中
        # 则增加看不到它的对数概率
        # 也就是log(1 - 看到它的概率)
        else:
            log_prob_if_spam += math.log(1.0 - prob_if_spam)
            log_prob_if_not_spam += math.log(1.0 - prob_if_not_spam)

    prob_if_spam = math.exp(log_prob_if_spam)
    prob_if_not_spam = math.exp(log_prob_if_not_spam)
    return prob_if_spam / (prob_if_spam + prob_if_not_spam)
```

将上面的代码结合起来，就得到了我们的朴素贝叶斯分类器：

```
class NaiveBayesClassifier:

    def __init__(self, k=0.5):
        self.k = k
        self.word_probs = []

    def train(self, training_set):

        # 对垃圾邮件和非垃圾邮件计数
        num_spams = len([is_spam
                         for message, is_spam in training_set
                         if is_spam])
        num_non_spams = len(training_set) - num_spams

        # 通过"pipeline"运行训练数据
        word_counts = count_words(training_set)
        self.word_probs = word_probabilities(word_counts,
                                             num_spams,
                                             num_non_spams,
                                             self.k)

    def classify(self, message):
        return spam_probability(self.word_probs, message)
```

13.4　测试模型

SpamAssassin 垃圾邮件公共语料库（https://spamassassin.apache.org/publiccorpus/）是一个非常不错（尽管有点老）的数据集。我们将考察其中前缀为 20021010 的文件。在 Windows 系统上，你可能需要用到类似 7-Zip（http://www.7-

zip.org/）的压缩软件来解压和提取文件。

提取数据之后（例如提取到 C:\spam 目录下面），你会看到 3 个文件夹：spam、easy_ham 和 hard_ham。每个文件夹中都存放了许多电子邮件，每封邮件都单独存放于一个文件之中。为简单起见，我们只检测每封邮件的主题行。

那么，我们应如何识别主题呢？通过观察可以发现，这些文件似乎都是以“Subject：”开头的。因此，我们可以利用下面的代码来识别主题内容：

```
import glob, re

# 把路径修改为你存放文件的那个
path = r"C:\spam\*\*"

data = []

# glob.glob会返回每一个与通配路径所匹配的文件名
for fn in glob.glob(path):
    is_spam = "ham" not in fn

    with open(fn,'r') as file:
        for line in file:
            if line.startswith("Subject:"):
                # 移除开头的"Subject: "，保留其余内容
                subject = re.sub(r"^Subject: ", "", line).strip()
                data.append((subject, is_spam))
```

好了，现在我们把数据分为训练数据和测试数据，然后开始建立分类器：

```
random.seed(0)        # 这样你能得到与我相同的答案
train_data, test_data = split_data(data, 0.75)

classifier = NaiveBayesClassifier()
classifier.train(train_data)
```

然后，我们可以检查一下模型的效果如何：

```
# 三个元素 (主题，确实是垃圾邮件，预测为垃圾邮件的概率)
classified = [(subject, is_spam, classifier.classify(subject))
              for subject, is_spam in test_data]

# 假设spam_probability > 0.5对应的是预测为垃圾邮件
```

```
# 对(actual is_spam, predicted is_spam)的组合计数
counts = Counter((is_spam, spam_probability > 0.5)
                 for _, is_spam, spam_probability in classified)
```

结果显示，真阳性（即垃圾邮件被分类为 spam）有 101 例，假阳性（即正常邮件被分类为 spam）有 33 例，真阴性（即正常邮件被分类为 ham）有 704 例，以及假阴性（即垃圾邮件被分类为 ham）有 38 例。也就是说，算法的查准率是 101 / (101 + 33) = 75%，查全率是 101 / (101 + 38) = 73%，对于如此简单的一个模型来说，这样的结果已经不错了。

由此也引出了一个有趣的问题，到底哪些邮件最容易被错误分类呢？请看下面的代码：

```
# 根据spam_probability从最小到最大排序
classified.sort(key=lambda row: row[2])

# 非垃圾邮件被预测为垃圾邮件的最高概率
spammiest_hams = filter(lambda row: not row[1], classified)[-5:]

# 垃圾邮件被预测为垃圾邮件的最低概率
hammiest_spams = filter(lambda row: row[1], classified)[:5]
```

这两种最容易被误判为垃圾邮件的正常邮件都含有单词 needed（它在垃圾邮件中出现的概率要高 77 倍）、insurance（它在垃圾邮件中出现的概率要高 30 倍）和 important（它在垃圾邮件中出现的概率要高 10 倍）。

最容易误判为正常邮件的垃圾邮件的标题都太短（“Re: girls”），以至于难以判断；排行第二的容易误判为正常邮件的垃圾邮件是信用卡邀约邮件，因为相关的词大多尚未被收录到训练集中。

同样，我们也可以看出现哪些词最容易被误判为垃圾邮件，具体代码如下所示：

```
def p_spam_given_word(word_prob):
    """uses bayes's theorem to compute p(spam | message contains word)"""

    # word_prob是由word_probabilities生成的三元素中的一个
    word, prob_if_spam, prob_if_not_spam = word_prob
    return prob_if_spam / (prob_if_spam + prob_if_not_spam)

words = sorted(classifier.word_probs, key=p_spam_given_word)
```

```
spammiest_words = words[-5:]
hammiest_words = words[:5]
```

最容易被判定为垃圾邮件的单词包括 money、systemworks、rates、sale 以及 year，这些似乎都与忽悠人买东西有关。而最容易被判定为正常邮件的单词有 spambayes、users、razor、zzzzteana 和 sadev，其中大部分好像都与阻止垃圾邮件相关，这比较奇怪。

那么，我们该如何改进性能呢？一个显而易见的方法是设法获取更多的训练数据。此外，还有许多可以改善模型本身的方法。大家不妨尝试以下具体的方法。

- 考察邮件内容，而不是仅仅考察邮件主题。你还必须仔细考虑邮件开头的处理方式。

- 我们的分类器考虑了训练集中包含的所有单词，即使该单词仅仅出现过一次。修改分类器，让它接受一个可选阈值 `min_count`，并且如果某个单词在训练集中出现次数少于阈值则不予考虑。

- 标记赋予器缺乏相似词（例如 cheap 和 cheapest）的概念。修改分类器，使其接受一个可选的词干分析器（`stemmer`）函数来找出单词对应的同类词。下面我们以一个非常简单的词干分析器函数为例进行介绍：

  ```
  def drop_final_s(word):
      return re.sub("s$", "", word)
  ```

  自己创建一个非常好用的词干分析器函数非常困难，所以人们通常使用现成的波特词干器（Porter Stemmer，http://tartarus.org/martin/PorterStemmer/）。

- 虽然我们的特征都是采取了“含有单词 w_i 的邮件”的形式，但这不代表必须采取这种形式。在我们的代码实现中，也可以添加额外的特征，比如“含有一个数字的邮件”。为此，可以创建类似 contains:number 这样的伪标记，然后修改标记赋予器（`tokenizer`），让它在适当的时候放出这些伪标记。

13.5　延伸学习

- Paul Graham 的“A Plan for Spam”（http://www.paulgraham.com/spam.html）和“Better Bayesian Filtering”（http://www.paulgraham.com/better.html）这两篇文章对如何打造垃圾 邮件过滤器提供了更加有趣而深入的介绍。

- `scikit-learn` 库（http://scikit-learn.org/stable/modules/naive_bayes.html）提供了一个名为 `BernoulliNB` 的模型，也实现了与本章介绍的相同的朴素贝叶斯算法以及基于该算法的其他变种。

第 14 章　简单线性回归

艺术就像道德，需要在某个地方画线。

——吉尔伯特 · 切斯特顿

在第 5 章中，我们曾经使用相关函数 `correlation` 来衡量两个变量之间的线性关系的强度。对于大多数应用来说，仅仅知道存在这样的线性关系是远远不够的。如果我们希望了解这种关系的性质，可以使用简单线性回归。

14.1　模型

别忘了，我们正在探讨的是 DataSciencester 用户的朋友数量与其每天花在该网站上的时间之间的关系。我们假设，你已经说服自己结交的朋友越多会导致人们花在网站上的时间越长。

这时，参与部的副总要你建立一个模型来描述这种关系。既然你发现有很强的线性关系，那么自然就要从线性模型开始着手了。准确地说，假设有常数 α（alpha）和 β（beta），使得：

$$y_i = \beta x_i + \alpha + \varepsilon_i$$

其中，y_i 是用户 i 每天花在网站上的分钟数，x_i 是用户 i 已有的朋友数，而 ε_i 是误差项，用来表示这个简单模型没有考虑到的其他因素，当然，误差项越小越好。

只要我们求出 `alpha` 和 `beta`，就能轻松通过下列公式来进行预测了：

```
def predict(alpha, beta, x_i):
    return beta * x_i + alpha
```

那么，该如何选择 `alpha` 和 `beta` 呢？实际上，只要任意选定的 `alpha` 和 `beta` 值，对于每个输入 `x_i`，都能得到一个预测的输出值。由于知道实际输出值 `y_i`，因此可以计算它们的误差：

```
def error(alpha, beta, x_i, y_i):
```

```
    """the error from predicting beta * x_i + alpha
    when the actual value is y_i"""
    return y_i - predict(alpha, beta, x_i)
```

实际上，我们真正想知道的是整个数据集的总体误差情况。不过，我们不能简单地将各个误差加起来，这是因为，如果 x_1 预测得太高，而 x_2 预测得太低，那么它们的误差加在一起就会相互抵消了。

因此，我们要对误差的平方求和：

```
def sum_of_squared_errors(alpha, beta, x, y):
    return sum(error(alpha, beta, x_i, y_i) ** 2
               for x_i, y_i in zip(x, y))
```

我们可以通过最小二乘法来选择 alpha 和 beta，以使 sum_of_squared_errors 尽可能小。

利用微积分（或单调乏味的代数），我们就可以求出令误差最小化的 alpha 和 beta 了，具体代码如下所示：

```
def least_squares_fit(x, y):
    """given training values for x and y,
    find the least-squares values of alpha and beta"""
    beta = correlation(x, y) * standard_deviation(y) / standard_deviation
    alpha = mean(y) - beta * mean(x)
    return alpha, beta
```

不要忙着进行严格的数学推导，让我们先想想为什么这可能是一个合理的解决方案。实际上，选定 alpha 后，只要给出自变量 x 的平均值，我们就能预测因变量 y 的平均值。

选定了 beta，就意味着输入值每增加 standard_deviation(x)，预测值就会增加 correlation(x, y) * standard_deviation(y)。就本例来说，如果 x 和 y 完全相关，则 x 每增加一个标准偏差，预测值就会增加 y 的一个标准偏差。当它们完全负相关的时候，预测值会随着 x 的增加而减小。当它们的相关性为 0 时，beta 为 0，这意味着 x 的变化根本不会对预测值产生影响。

下面我们使用第 5 章中的异常值数据来计算这两个值：

```
alpha, beta = least_squares_fit(num_friends_good, daily_minutes_good)
```

计算结果是，alpha ＝ 22.95，beta ＝ 0.903。因此，根据我们的模型来看，具有 `n` 个好友的用户每天会在这个网站上花 `22.95 + n * 0.903` 分钟。同时，对于在 DataSciencester 上面没有朋友的用户来说，他们每天仍然会花 23 分钟泡在这个网站上。此外，用户每增加一个朋友，每天花费在这个网站上的时间就会多出一分钟左右。在图 14-1 中，我们绘制了该模型的预测线，从中可以看出模型的预测与观测数据的拟合效果。

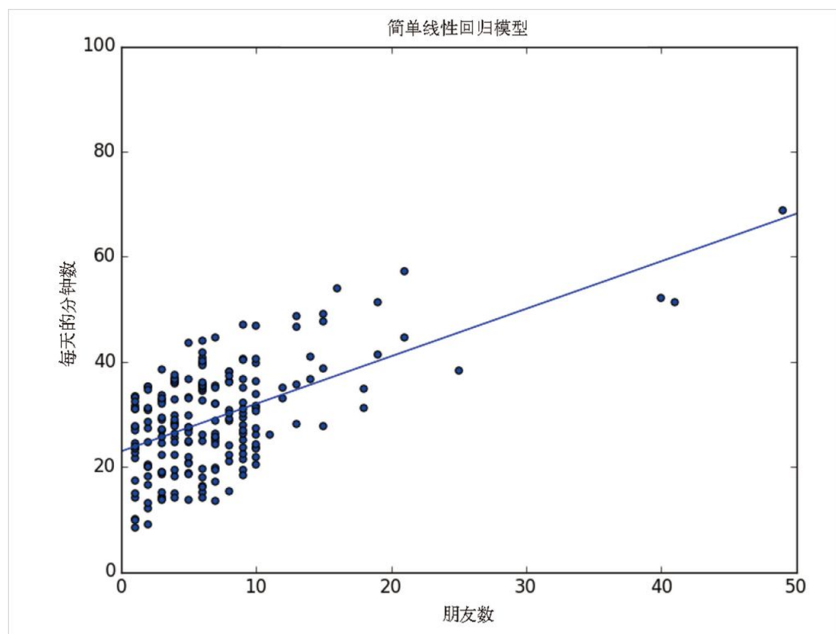


图 **14-1**：简单线性模型

当然，仅仅依靠目测是不够的，我们需要一个更好的指标来评估模型对数据的拟合效果。一个常见的指标是决定系数（coefficient of determination）或 R 平方，用来表示纳入模型的自变量引起的变动占总变动的百分比：

```
def total_sum_of_squares(y):
    """the total squared variation of y_i's from their mean"""
    return sum(v ** 2 for v in de_mean(y))

def r_squared(alpha, beta, x, y):
```

```
    """the fraction of variation in y captured by the model, which equals
    1 - the fraction of variation in y not captured by the model"""

    return 1.0 - (sum_of_squared_errors(alpha, beta, x, y) /
                  total_sum_of_squares(y))

r_squared(alpha, beta, num_friends_good, daily_minutes_good)      # 0.329
```

现在，我们已经选取了令预测误差平方和最小化的 alpha 和 beta。我们选择的是一个“总是预测 mean(y)”的线性模型（即 alpha = mean(y) 且 beta = 0），模型误差平方和正好等于其平方总和。这就意味着拟合优度（R 平方）的值为 0，表明模型（显然，在这种情况下）几乎只能预测平均值。

很明显，最小二乘模型最差的时候，就是误差的平方和最大为平方总和的时候，也是 R 平方最小为 0 的时候。同时，因为误差的平方和至少为 0，所以 R 平方至多为 1。

R 平方的值越大，说明模型对数据的拟合度越高。在这里，R 平方的值为 0.329，说明模型对这些数据的拟合度不是很高，显然还有没考虑到的其他因素在起作用。

14.2　利用梯度下降法

如果我们记 theta = [alpha, beta]，那么也可以通过梯度下降法来求参数：

```
def squared_error(x_i, y_i, theta):
    alpha, beta = theta
    return error(alpha, beta, x_i, y_i) ** 2

def squared_error_gradient(x_i, y_i, theta):
    alpha, beta = theta
    return [-2 * error(alpha, beta, x_i, y_i),       # alpha偏导数
            -2 * error(alpha, beta, x_i, y_i) * x_i] # beta偏导数

# 选择一个随机值作为开始
random.seed(0)
theta = [random.random(), random.random()]
alpha, beta = minimize_stochastic(squared_error,
                                  squared_error_gradient,
                                  num_friends_good,
                                  daily_minutes_good,
                                  theta,
                                  0.0001)
```

```
print alpha, beta
```

使用相同的数据，我们得到 alpha = 22.93，beta = 0.905，这与精确的答案非常接近。

14.3　最大似然估计

我们为什么会选择最小二乘法呢？其中一个原因就是最大似然估计（maximum likelihood estimation）。假设我们的数据样本 $v_1, \cdots, v_n$ 服从由未知参数 θ 确定的概率分布：

$$p(v_1, \cdots, v_n | \theta)$$

虽然我们不知道 θ，但是可以回过头来通过给定样本与 θ 的相似度来考量这个参数：

$$L(\theta | v_1, \cdots, v_n)$$

按照这种方法，θ 最可能的值就是最大化这个似然函数的值，即能够以最高概率产生观测数据的值。在具有概率分布函数而非概率密度函数的连续分布的情况下，我们也可以做同样的事情。

再回到回归这个话题。对于简单回归模型来说，通常假设回归误差是呈正态分布的，其均值为 0，并且已知标准偏差 σ。如果是这样的话，那么就可以通过下面的似然函数来描述 α 和 β 产生 `(x_i, y_i)` 的可能性大小了：

$$L(\alpha, \beta | x_i, y_i, \sigma) = \frac{1}{\sqrt{2\pi\sigma}} \exp(-(y_i - \alpha - \beta x_i)^2 / 2\sigma^2)$$

由于待估计的参数产生整个数据集的可能性为产生各个数据的可能性之积，因此令误差的平方和最小的 `alpha` 和 `beta` 最有可能是我们所求的。换句话说，在这种情况下（包括这些假设），最小化误差的平方和等价于最大化产生观测数据的可能性。

14.4　延伸学习

请继续往下阅读第 15 章介绍的多重回归分析！

第 15 章　多重回归分析

我不会盯着一个问题看，并往其中添加无用的变量。

——比尔 · 帕塞尔斯

虽然副总对你的预测模型很满意，但是他认为你还可以做得更好。为此，你收集了额外的数据：对于每一个用户，你不仅了解他每天工作多少小时，同时还调查了他是否拥有博士学位。你希望通过这些补充资料来改进模型。

因此，你提出了一个带有更多自变量的线性模型：

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \beta_2 \text{work hours} + \beta_3 \text{phd} + \varepsilon$$

显然，用户是否拥有博士学位并非一个数值问题，但如同第 11 章所提到的，我们可以引入一个虚拟变量，当这个变量等于 1 的时候，表示用户拥有博士学位，反之则表示没有博士学位，这样就能像其他变量一样将其视为一个数值了。

15.1　模型

回想一下，我们在第 14 章中所拟合的模型形式如下所示：

$$y_i = \alpha + \beta x_i + \varepsilon_i$$

现在，如果每个输入 x_i 不再是单个数字，而是一个由 k 个数字 $x_i 1, \cdots, x_i k$ 组成的向量，那么我们的多重回归模型则应该为：

$$y_i = \alpha + \beta_1 x_{i1} + \cdots + \beta_k x_{ik} + \varepsilon_i$$

对于多重回归分析来说，参数向量通常称为 β。我们希望这个向量也包括一个常数项，为此，只要向我们的数据中添加一列即可：

```
beta = [alpha, beta_1, ..., beta_k]
```

同时：

```

```

```
x_i = [1, x_i1, ..., x_ik]
```

那么，我们的模型可以用下列函数实现：

```
def predict(x_i, beta):
    """assumes that the first element of each x_i is 1"""
    return dot(x_i, beta)
```

就本例而言，自变量 x 是一个向量型列表，每个列表元素如下所示：

```
[1,     # 常数项
 49,    # 朋友数
 4,     # 每日工作时长
 0]     # 没有博士学位
```

15.2　最小二乘模型的进一步假设

对于我们的模型（以及我们的解决方案）来说，需要添加另外两个假设，才能够言之有理。

第一个假设是 x 的各个列是线性无关的，即任何一列绝对不会是其他列的加权和。如果这个假设不成立，则无法估计 beta。为了了解极端的情形，我们可以想象数据中有一个额外的字段 num_acquaintances，并且对于每一个用户来说它都等于 num_friends。

那么，对于任何 beta，如果 num_friends 的系数增加了某个数值，而 num_acquaintances 的系数同时减小相同数值的话，那么模型的预测就会保持不变。也就是说，我们根本就没有办法确定 num_friends 的系数。（通常来说，对于这个假设的违背情况一般不会这么明显。）

第二个重要的假设是 x 的各列与误差 ε 无关。如果这个假设不成立，对于 beta 的估计就会出现系统性的错误。

比如，在第 14 章中，我们建立的模型的预测结果为，用户每增加一个朋友，每天花在网站上的时间就会多出 0.90 分钟。

想象一下，还有下列情况。

- 工作时间越长的人在网站上花的时间越少。

- 朋友更多的人倾向于工作更长时间。

也就是说，假设“实际的”模型为：

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \beta_2 \text{work hours} + \varepsilon$$

并且工作时间和朋友数量是正相关的。那样的话，当我们最小化单变量模型的误差时：

$$\text{minutes} = \alpha + \beta_1 \text{friends} + \varepsilon$$

我们会低估 β_1。

考虑一下，如果这个单变量模型已知 β_1 的“实际”值，那么这时再用它来预测将会如何。（亦即，这个值来自令误差最小化的“实际”模型。）这时候，对工作时间比较长的用户来说，产生的预测值往往太小；对工作时间比较短的用户来说，产生的预测值往往又过大，这是因为 $\beta_2 > 0$，但是我们“忘了”把它考虑进去。由于工作时间与朋友的数量是呈正相关的，这就意味着对于朋友数量较多的用户来说，模型给出的预测值往往太小；对于朋友数量较少的用户来说，模型给出的预测值往往太大。这样做的结果是，我们可以通过降低 β_1 的估计值来减少（单变量模型）的误差，即误差最小化的 β_1 是小于其“实际”值的。也就是说，在这种情况下，单变量的最小二乘解偏向于低估 β_1。一般而言，当自变量具有与此类似的误差时，我们的最小二乘解给出的 β 是有偏估计。

15.3　拟合模型

就像对简单线性模型所做的那样，我们这里也需要寻找一个能够最小化误差的平方和的 `beta`。要想以手动方式找到一个精确的解可不是一件容易的事，因此，我们转而求助于梯度下降。下面我们首先创建一个待最小化的误差函数。对于随机梯度下降来说，我们只需要单次预测对应的平方误差：

```
def error(x_i, y_i, beta):
    return y_i - predict(x_i, beta)

def squared_error(x_i, y_i, beta):
    return error(x_i, y_i, beta) ** 2
```

如果你熟悉微积分，可以通过下面的方式进行计算：

```
def squared_error_gradient(x_i, y_i, beta):
    """the gradient (with respect to beta)
    corresponding to the ith squared error term"""
    return [-2 * x_ij * error(x_i, y_i, beta)
            for x_ij in x_i]
```

否则的话，就按照我说的来。

至此，我们就可以利用随机梯度下降法来寻找最优的 beta 了：

```
def estimate_beta(x, y):
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(squared_error,
                               squared_error_gradient,
                               x, y,
                               beta_initial,
                               0.001)

random.seed(0)
beta = estimate_beta(x, daily_minutes_good) # [30.63, 0.972, -1.868, 0.91
```

这样的话，我们的模型就变成了：

$$\text{minutes} = 30.63 + 0.972 \text{ friends} - 1.868 \text{ work hours} + 0.911 \text{ phd}$$

15.4　解释模型

你应该把模型的系数看作在其他条件相同的情况下每个因素的影响力的大小。在其他条件相同的情况下，每增加一个朋友，每天花在网站上的时间就会多出一分钟。在其他条件相同的情况下，用户在工作日每多工作一个小时，每天花在网站上的时间就会减少两分钟。在其他条件相同的情况下，拥有博士学位的用户每天用在网站上的时间会多出一分钟。

但是，它没有（直接）反映变量之间的任何相互作用。较之于朋友较少的人而言，工作时间对朋友较多的人的影响很可能是不一样的，而这个模型并没有捕捉到这一点。要想处理这种情况，一种方法是引入一个新变量，即“朋友数量”与“工作时间”之积。这样实际上就使得“工作时间”的系数可以随着朋友数量的增加而增加（或减少）。

还有一种可能，就是朋友越多，花在网站上的时间就越多，但是达到一个上限之后，更多的朋友反而会导致花在网站上的时间变少。（或许是因为朋友太多了反而上网体验会很糟糕？）我们可以设法让模型捕获到这一点，方法是添加另一个变量，即朋友数量的平方。

一旦我们开始添加变量，我们就需要考虑它们的系数“问题”。对于添加的乘积、对数、二次幂以及更高次幂来说，其数量上是没有限制的。

15.5　拟合优度

现在，我们再来看看 R 的平方值，目前已经升至 0.68 了：

```
def multiple_r_squared(x, y, beta):
    sum_of_squared_errors = sum(error(x_i, y_i, beta) ** 2
                                for x_i, y_i in zip(x, y))
    return 1.0 - sum_of_squared_errors / total_sum_of_squares(y)
```

但是不要忘了，只要向回归模型中添加新的变量就必然导致 R 的平方变大。归根结底，前面的简单回归模型只是这里的多重回归模型的特例而已，即“工作时间”和“博士”这两列的系数都等于 0。因此，最优的多元回归模型，其误差一定不会高于简单回归模型。

因此，对于多重回归分析而言，我们还要考察系数的标准误差，即衡量每个 β_i 的估计值的可靠程度。

总的来说，回归模型通常能够很好地拟合我们的数据，但是，如果某些自变量是相关的（或不相关的），那么其系数就未必有多大的意义了。

对于这些误差，传统的度量方法通常都带有一个前提假设，即误差 ε_i 是独立的正态随机变量，其平均值为 0，标准偏差为 σ（未知）。那样的话，我们（或者说我们的统计软件）就可以使用线性代数来确定每个系数的标准误差了。这个误差越大，说明模型的系数越不靠谱。令人遗憾的是，我们不打算从零开始介绍这类线性代数。

15.6　题外话：Bootstrap

假设我们有一个含有 n 个数据点的样本，并且这些点是按照某种（我们不知道的）概率分布生成的：

```
data = get_sample(num_points=n)
```

在第 5 章中，我们曾经编写了一个计算观测数据中位数的函数，现在拿它来估算该分布本身的中位数。

但是，我们该如何了解这些估计值的可靠性呢？如果样品中所有的数据都非常接近 100，则实际的中位数很可能也非常接近 100。如果样本中一半左右的数据接近 0，而另一半则接近 200，那么我们就很难确信中位数到底接近多少。

如果我们能够不断获得新的样本，那么就可以计算出每个新样本的中位数，并观察这些中位数的分布情况。但是，一般这是不现实的。相反，我们可以利用 Bootstrap 来获得新的数据集，即选择 n 个数据点并用原来的数据将其替换，然后计算合成的数据集的中位数：

```
def bootstrap_sample(data):
    """randomly samples len(data) elements with replacement"""
    return [random.choice(data) for _ in data]

def bootstrap_statistic(data, stats_fn, num_samples):
    """evaluates stats_fn on num_samples bootstrap samples from data"""
    return [stats_fn(bootstrap_sample(data))
            for _ in range(num_samples)]
```

例如，考虑下列两个数据集：

```
# 101个点都非常接近100
close_to_100 = [99.5 + random.random() for _ in range(101)]

# 101个点钟，50个接近0，50个接近200
far_from_100 = ([99.5 + random.random()] +
                [random.random() for _ in range(50)] +
                [200 + random.random() for _ in range(50)])
```

如果你计算每个数据集的中位数，会发现它们都非常接近 100。然而，如果你考察下面的语句：

```
bootstrap_statistic(close_to_100, median, 100)
```

大部分情况下你看到的数字确实非常接近 100。然而，如果你考察下面的语句：

```
bootstrap_statistic(far_from_100, median, 100)
```

你会发现，不仅有许多数字接近 0，而且还有许多数字接近 200。

第一组中位数的 standard_deviation 接近 0，而第二组中位数的 standard_deviation 接近 100。（这种极端的情况通过人工检查数据很容易弄清楚，但一般情况下都不是真的。）

15.7 回归系数的标准误差

我们可以采用同样的方法来估计回归系数的标准误差。我们可以对数据重复采用 bootstrap_sample 样本，并根据这些样本估算 beta。如果某个自变量（如 num_friends）的系数在各个样本上变化不大，那么就可以确信我们的估计是比较严密的。如果这个系数随着样本的不同而起伏较大，那么我们就不能完全相信我们的估计。

唯一需要说明的是，采样前，我们需要把数据 x 和数据 y 放到一起（用 zip），以确保对自变量和因变量一起进行采样。这就意味着 bootstrap_sample 将返回一个由 (x_i, y_i) 数据对组成的列表，因此我们需要将其重新组合成一个 x_sample 和一个 y_sample：

```
def estimate_sample_beta(sample):
    """sample is a list of pairs (x_i, y_i)"""
    x_sample, y_sample = zip(*sample) # 魔法般的解压方式
    return estimate_beta(x_sample, y_sample)

random.seed(0) # 所以你得到的结果与我的一样

bootstrap_betas = bootstrap_statistic(zip(x, daily_minutes_good),
                                      estimate_sample_beta,
                                      100)
```

之后，我们就可以估算每个系数的标准偏差了：

```
bootstrap_standard_errors = [
    standard_deviation([beta[i] for beta in bootstrap_betas])
    for i in range(4)]

# [1.174,    # 常数项，       实际误差 = 1.19
#  0.079,    # num_friends， 实际误差 = 0.080
#  0.131,    # unemployed,    实际误差 = 0.127
```

```
#  0.990]    # phd,          实际误差 = 0.998
```

我们可以使用它们来检验诸如“β_i 等于 0 吗？”之类的假设。在满足 $\beta_i = 0$（以及与 ε_i 分布 有关的其他假设）的条件下，则有：

$$t_j = \hat{\beta}_j / \hat{\sigma}_j$$

也就是说，这个统计量等于我们估算的 β_j 除以估算的其标准误差，它符合具有“n-k 个自由度”的学生的 t 分布（Student's t-distribution）。

如果我们有一个 `students_t_cdf` 函数，那么就可以计算每个最小二乘系数的 p 值，从而指出实际的系数为 0 时观察到这个值的可能性有多大。令人遗憾的是，实际上我们没有这样的函数。（虽然我们不想从头做起。）

然而，随着自由度变大，t 分布越接近标准正态分布。在这种情况下，即 n 比 k 大得多的情况下，我们便可以使用 `normal_cdf` 了，并且我们觉得它效果还不错：

```
def p_value(beta_hat_j, sigma_hat_j):
    if beta_hat_j > 0:
        # 如果系数是正的，则我们需要对
        # 看见一个更大的值的概率做两次计算
        return 2 * (1 - normal_cdf(beta_hat_j / sigma_hat_j))
    else:
        # 否则看见一个更小值的概率乘以2
        return 2 * normal_cdf(beta_hat_j / sigma_hat_j)

p_value(30.63, 1.174)    # ~0   (常数项)
p_value(0.972, 0.079)    # ~0   (num_friends)
p_value(-1.868, 0.131)   # ~0   (work_hours)
p_value(0.911, 0.990)    # 0.36 (phd)
```

（在其他情况下，我们很可能会使用一个知道如何计算 t 分布和精确的标准误差的统计软件。）

虽然大多数系数的 p 值都非常小（但非 0 值），但是“博士学位”的系数与零没有“显著”区别，也就是说“博士学位”的系数很可能是随机的，无意义的。

在对回归分析要求更加精细的情形下，你可能需要对数据的各种假设

进行更加细致的测试，比如“至少有一个 β_j 是非 0 值”，或者“β_1 等于 β_2 且 β_3 等于 β_4”等，以便进行 F 测试，但是这些内容已经超出了本书的讨论范围。

15.8　正则化

在实践中，线性回归经常需要处理具有很多变量的数据集，这时就需要用到另外两个技巧。首先，涉及的变量越多，模型越容易对训练集产生过拟合现象。其次，非零系数越多，越难以搞清楚它们的意义。如果我们的目标是解释某些现象，一个只考虑三方面因素的稀疏型模型通常要比涉及数百个因素的模型要更好一些。

正则化是指给误差项添加一个惩罚项，并且该惩罚项会随着 beta 的增大而增大。然后，我们开始设法将误差项和惩罚项的组合值最小化。因此，惩罚项越大，就越能防止系数过大。

例如，在岭回归（ridge regression）中，我们添加了一个与 beta_i 的平方之和成正比的惩罚项。（当然，我们一般不会惩罚 beta_0，因为它是个常数项。）

```
# alpha是一个*超参数*，用来控制惩罚的程度
# 它有时被叫作"lambda"，但这在Python中另有所指
def ridge_penalty(beta, alpha):
  return alpha * dot(beta[1:], beta[1:])

def squared_error_ridge(x_i, y_i, beta, alpha):
    """estimate error plus ridge penalty on beta"""
    return error(x_i, y_i, beta) ** 2 + ridge_penalty(beta, alpha)
```

之后你可以按通常的方法插入梯度下降：

```
def ridge_penalty_gradient(beta, alpha):
    """gradient of just the ridge penalty"""
    return [0] + [2 * alpha * beta_j for beta_j in beta[1:]]

def squared_error_ridge_gradient(x_i, y_i, beta, alpha):
    """the gradient corresponding to the ith squared error term
    including the ridge penalty"""
    return vector_add(squared_error_gradient(x_i, y_i, beta),
                      ridge_penalty_gradient(beta, alpha))

def estimate_beta_ridge(x, y, alpha):
    """use gradient descent to fit a ridge regression
    with penalty alpha"""
```

```
    beta_initial = [random.random() for x_i in x[0]]
    return minimize_stochastic(partial(squared_error_ridge, alpha=alpha),
                               partial(squared_error_ridge_gradient,
                                       alpha=alpha),
                               x, y,
                               beta_initial,
                               0.001)
```

如果令 alpha 为 0，则根本不会实施任何惩罚，这时得到的结果跟前面一样：

```
random.seed(0)
beta_0 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.0)
# [30.6, 0.97, -1.87, 0.91]
dot(beta_0[1:], beta_0[1:]) # 5.26
multiple_r_squared(x, daily_minutes_good, beta_0) # 0.680
```

随着 alpha 的增大，拟合优度会变差，但是 beta 会变小：

```
beta_0_01 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.01)
# [30.6, 0.97, -1.86, 0.89]
dot(beta_0_01[1:], beta_0_01[1:])  # 5.19
multiple_r_squared(x, daily_minutes_good, beta_0_01)  # 0.680

beta_0_1 = estimate_beta_ridge(x, daily_minutes_good, alpha=0.1)
# [30.8, 0.95, -1.84, 0.54]
dot(beta_0_1[1:], beta_0_1[1:])  # 4.60
multiple_r_squared(x, daily_minutes_good, beta_0_1)  # 0.680

beta_1 = estimate_beta_ridge(x, daily_minutes_good, alpha=1)
# [30.7, 0.90, -1.69, 0.085]
dot(beta_1[1:], beta_1[1:])  # 3.69
multiple_r_squared(x, daily_minutes_good, beta_1)  # 0.676

beta_10 = estimate_beta_ridge(x, daily_minutes_good, alpha=10)
# [28.3, 0.72, -0.91, -0.017]
dot(beta_10[1:], beta_10[1:])  # 1.36
multiple_r_squared(x, daily_minutes_good, beta_10)  # 0.573
```

特别地，随着惩罚项的增大，“博士学位”的系数会变成 0，这与我们之前的结果是一致的，即它与 0 没有显著区别。



在利用这个方法之前，通常需要调整数据的规模。因为即使是同一个模型，如果将几年数据一下变为几百年的数据，那么它的最小二乘法系数就会增加

上百倍，那样得到的惩罚肯定也会骤增。

还有一个方法是 lasso 回归，它用的惩罚方式如下所示：

```
def lasso_penalty(beta, alpha):
    return alpha * sum(abs(beta_i) for beta_i in beta[1:])
```

总的说来，岭回归的惩罚项会缩小系数，但是，lasso 的惩罚项却趋向于迫使系数变为 0 值，这使得它更适于学习稀疏模型。令人遗憾的是，它不适用于梯度下降法，这意味着我们将无法从头开始解决这个问题。

15.9　延伸学习

- 回归分析具有深厚而广阔的理论背景。要想了解这些背景理论，你需要阅读相应的教科书，至少也得阅读大量的维基百科文章。

- scikit-learn 的 linear_model 模块（http://scikit-learn.org/stable/modules/linear_model.html）提供了一个 `LinearRegression` 模型，它跟我们的模型颇为相近。此外，它还提供了 `Ridge` 回归和 `Lasso` 回归，以及其他类型的正则化算法。

- 另一个相关的 Python 模块是 Statsmodels（http://statsmodels.sourceforge.net/），它也包含了线性回归模型及许多其他内容。

第 16 章　逻辑回归

许多人说，天才和疯子之间只有一线之隔。但是我却认为，那不是一线之隔，而是天渊之别。

——比尔 · 贝利

在第 1 章中，我们简单介绍了如何预测哪些 DataSciencester 用户会成为付费用户。下面，我们将进一步考察这个问题。

16.1　问题

我们有一个 200 人的匿名数据集，内容包括每个用户的工资、作为数据科学家的工作年限以及是否愿意成为付费用户（图 16-1）。像往常一样，对于类别型变量，它们的取值要么是 0（非付费用户），要么是 1（付费用户）。

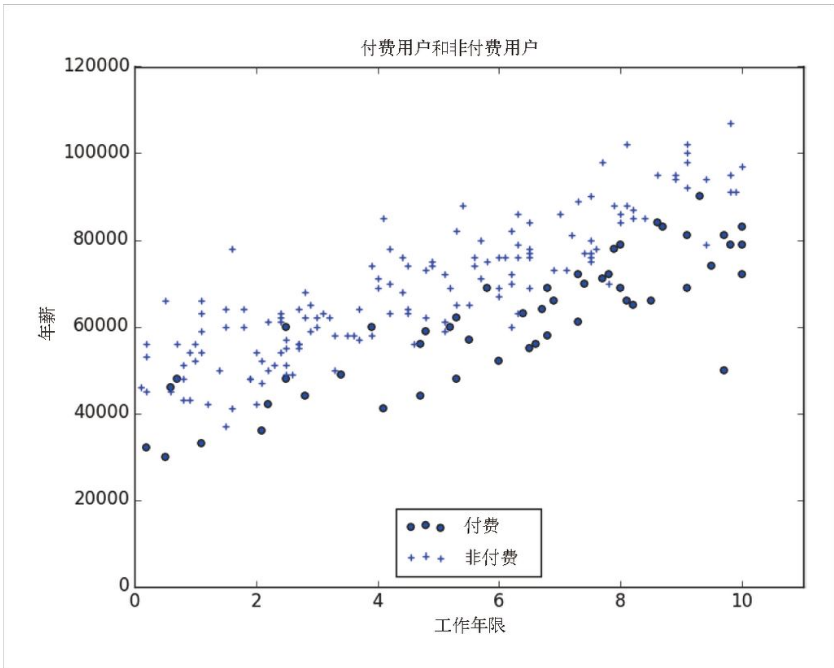


图 **16-1**：付费用户和非付费用户

像往常一样，我们的数据都是存放到矩阵中的，其中每一行都是一个列表 [experience, salary, paid_account]。下面，我们将其转换成所需要的格式：

```
x = [[1] + row[:2] for row in data]    # 每个元素都是[1, experience, salary]
y = [row[2] for row in data]           # 每个元素都是一个付费用户
```

很明显，第一步是使用线性回归来寻找最佳模型：

$$\text{paid account} = \beta_0 + \beta_1 \text{experience} + \beta_2 \text{salary} + \varepsilon$$

当然，在利用这种方式对这个问题进行建模方面，我们已经轻车熟路了。

结果如图 16-2 所示：

```
rescaled_x = rescale(x)
beta = estimate_beta(rescaled_x, y)  # [0.26, 0.43, -0.43]
predictions = [predict(x_i, beta) for x_i in rescaled_x]

plt.scatter(predictions, y)
plt.xlabel("predicted")
plt.ylabel("actual")
plt.show()
```

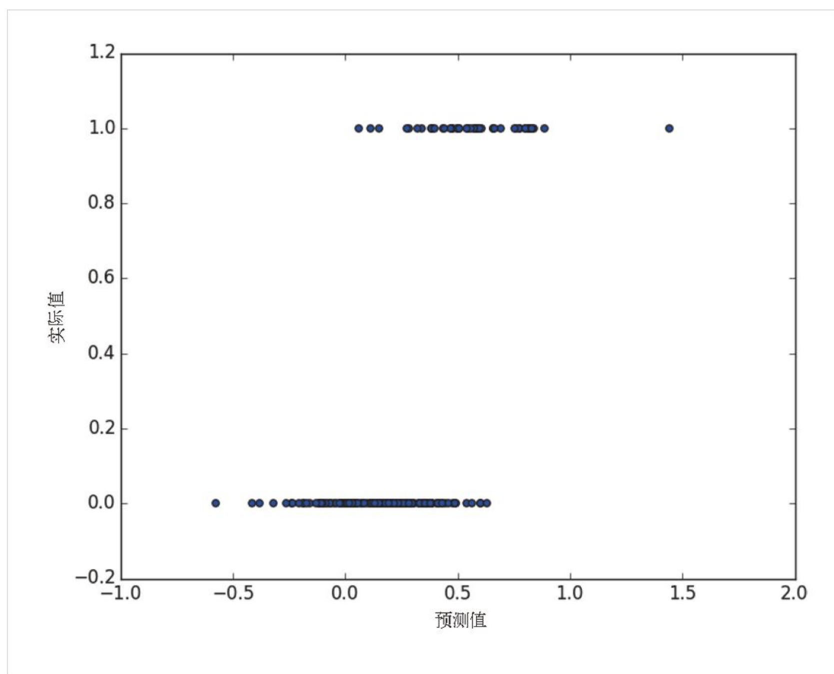


图 **16-2**：使用线性回归预测付费用户

但这种方法会导致两个紧密相关的问题。

- 我们希望输出结果为 0 或者 1，用来表明会员资格类型。如果输出值介于 0 和 1 之间的话，那也很好，因为我们可以将其解读为概率——如果输出值是 0.25，可以表示成为付费会员的可能性为 25%。但是，线性模型的输出值有时候会是非常大的正数乃至负数，这时候就不好解释了。实际上，这个例子中许多预测值是负数。

- 线性回归模型假定 x 的各列与误差不相关。但是这里 `experience` 一列的回归系数为 0.43，也就是说，数据科学家生涯越长，就越有可能成为付费用户。这意味着对于职业生涯较长的人，模型会输出一个很大的数值。但是我们都知道，有效值最大只能到 1，也就是说输出值越大（即数据科学家生涯越长），对应的误差项的负值也就越大。由于这个原因，我们对 `beta` 的估计是有偏的。

相反，我们期望的情况是这样的：如果 `dot(x_i, beta)` 的输出值是较大的正数，那么让它对应的概率接近 1；如果输出值是一个较大

的负数，那么让它对应的概率接近 0。为此，我们可以应用另一个函数来实现这种效果。

16.2　Logistic函数

对于逻辑回归来说，我们需要用到 Logistic 函数，其图像如图 16-3 所示：

```
def logistic(x):
    return 1.0 / (1 + math.exp(-x))
```

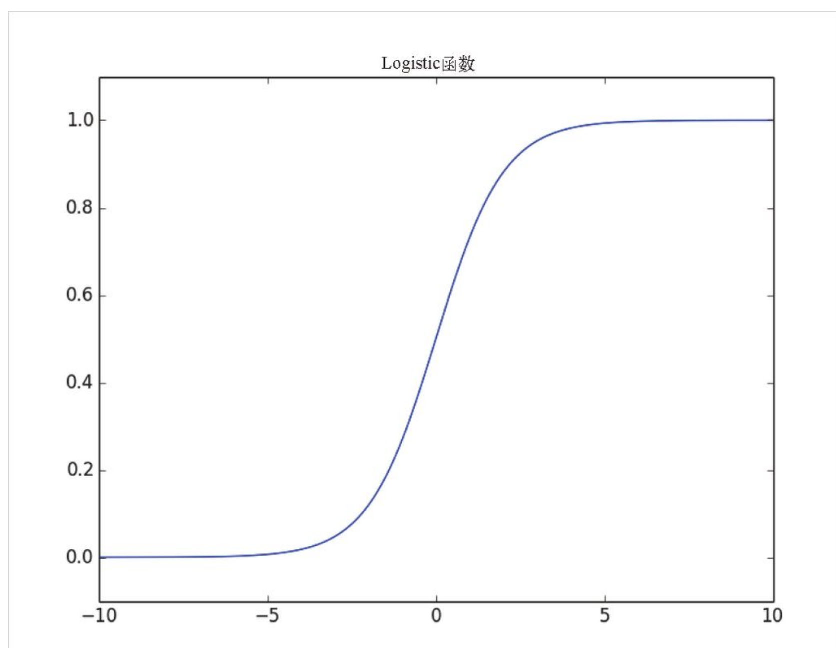


图 16-3：Logistic 函数

随着输入的数字变大且符号为正，它的输出就会越来越接近 1。随着输入的数字变大且符号为负，它的输出就会越来越接近 0。此外，这个函数还有一个非常好的属性，即其导数可以通过下列代码简单求出：

```
def logistic_prime(x):
    return logistic(x) * (1 - logistic(x))
```

这一点可以用于拟合模型：

$$y_i = f(x_i\beta) + \varepsilon_i$$

其中，f 表示 `logistic` 函数。

回想一下，对于线性回归来说，我们是通过最小化误差的平方之和的方式来拟合模型，最终选出令得到这些观测数据的可能性最大的 β。

但是，这里两者并不是等价的，所以我们直接使用梯度下降法来最大化似然。也就是说，我们需要计算似然函数及其梯度。

已知 β，我们的模型指出每个 y_i 等于 1 的概率为 $f(x_i\beta)$，等于 0 的概率为 $1-f(x_i\beta)$。

特别是，y_i 的概率密度函数为：

$$p(y_i|x_i,\beta) = f(x_i\beta)^{y_i}(1-f(x_i\beta))^{1-y_i}$$

如果 y_i 为 0，则这等同于：

$$1 - f(x_i\beta)$$

且如果 y_i 为 1，则等同于：

$$f(x_i\beta)$$

事实表明，最大化对数似然要更加简单一些：

$$\log L(\beta|x_i,y_i) = y_i \log f(x_i\beta) + (1-y_i)\log(1-f(x_i\beta))$$

由于对数函数是单调递增函数，所以任何能够最大化对数似然函数的 `beta` 必然也能最大化似然函数，反之亦然。

```
def logistic_log_likelihood_i(x_i, y_i, beta):
    if y_i == 1:
        return math.log(logistic(dot(x_i, beta)))
    else:
        return math.log(1 - logistic(dot(x_i, beta)))
```

如果我们假设各个数据点之间相互独立，那么整体的似然就是各个似然之积。换句话说，整体的对数似然就是各个对数似然之和：

```
def logistic_log_likelihood(x, y, beta):
    return sum(logistic_log_likelihood_i(x_i, y_i, beta)
               for x_i, y_i in zip(x, y))
```

利用少许微积分知识，我们就能求出梯度了：

```
def logistic_log_partial_ij(x_i, y_i, beta, j):
    """here i is the index of the data point,
    j the index of the derivative"""

    return (y_i - logistic(dot(x_i, beta))) * x_i[j]

def logistic_log_gradient_i(x_i, y_i, beta):
    """the gradient of the log likelihood
    corresponding to the ith data point"""

    return [logistic_log_partial_ij(x_i, y_i, beta, j)
            for j, _ in enumerate(beta)]

def logistic_log_gradient(x, y, beta):
    return reduce(vector_add,
                  [logistic_log_gradient_i(x_i, y_i, beta)
                   for x_i, y_i in zip(x,y)])
```

好了，到此为止我们已经万事俱备了。

16.3　应用模型

现在我们要将数据分为一个训练集和一个测试集：

```
random.seed(0)
x_train, x_test, y_train, y_test = train_test_split(rescaled_x, y, 0.33)

# 希望在训练数据集上最大化对数似然
fn = partial(logistic_log_likelihood, x_train, y_train)
gradient_fn = partial(logistic_log_gradient, x_train, y_train)

# 选取一个随机起始点
beta_0 = [random.random() for _ in range(3)]

# 使用梯度下降法实现最大化
beta_hat = maximize_batch(fn, gradient_fn, beta_0)
```

另外，你也可以使用随机梯度下降法：

```
beta_hat = maximize_stochastic(logistic_log_likelihood_i,
                               logistic_log_gradient_i,
                               x_train, y_train, beta_0)
```

无论使用哪种方式，我们都能得到大致如下的结果：

```
beta_hat = [-1.90, 4.05, -3.87]
```

这些数据是按照某些系数转换过来的，不过，我们还可以将其转换为原始数据：

```
beta_hat_unscaled = [7.61, 1.42, -0.000249]
```

令人遗憾的是，这些系数不如线性回归系数那样易于解释。在其他条件都相同的情况下，工作年限每增加一年，`logistic` 的输入就会增加 1.42。在其他条件都相同的情况下，年薪每增加 10 000 美元，`logistic` 的输入就会减去 2.49。

然而，输出的结果还会受到其他输入数据的影响。如果 `dot(beta, x_i)` 的值已经很大了（相当于概率接近 1），那么即使再增加的话，对概率也没有多大的影响了。如果它接近 0，那么即使稍微增加一点，对概率也会产生明显的影响。

也就是说，在其他条件相同的情况下，工作年限越多的人越有可能成为付费用户。同时，其他条件相同的情况下，年薪越高的人越不可能成为付费用户。（当我们将数据绘成图表的时候，这一点就会更明显。）

16.4　拟合优度

目前为止，我们还没有使用留出来的测试数据。下面来看看，如果我们预测成为付费用户的概率大于 0.5 的话会发生什么：

```
true_positives = false_positives = true_negatives = false_negatives = 0

for x_i, y_i in zip(x_test, y_test):
    predict = logistic(dot(beta_hat, x_i))

    if y_i == 1 and predict >= 0.5:    # TP: 是付费用户，且我们预测为是
        true_positives += 1
```

```
    elif y_i == 1:                              # FN: 是付费用户，且我们预测为否
        false_negatives += 1
    elif predict >= 0.5:                        # FP: 非付费用户，且我们预测为是
        false_positives += 1
    else:                                       # TN: 非付费用户，且我们预测为否
        true_negatives += 1

precision = true_positives / (true_positives + false_positives)
recall = true_positives / (true_positives + false_negatives)
```

这里的查准率为 93%（即每预测 100 次有 93 次是正确的），查全率为 82%（即每 100 个付费用户中，我们能够预测出 82 个人），这两项指标都相当不错。

在图 16-4 中，我们给出了系统的预测值与实际值的比较情况，图中表明该模型的表现非常好：

```
predictions = [logistic(dot(beta_hat, x_i)) for x_i in x_test]
plt.scatter(predictions, y_test)
plt.xlabel("predicted probability")
plt.ylabel("actual outcome")
plt.title("Logistic Regression Predicted vs. Actual")
plt.show()
```

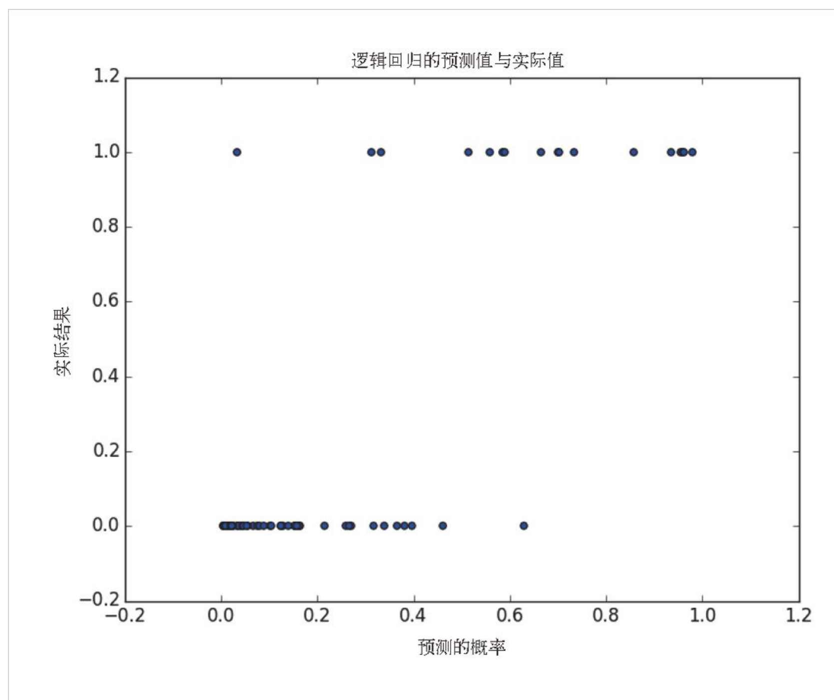


图 **16-4**：逻辑回归的预测值与实际值

16.5　支持向量机

`Dot(beta_hat, x_i)` 等于 0 的点就是我们的分类边界线，具体如图 16-5 所示。

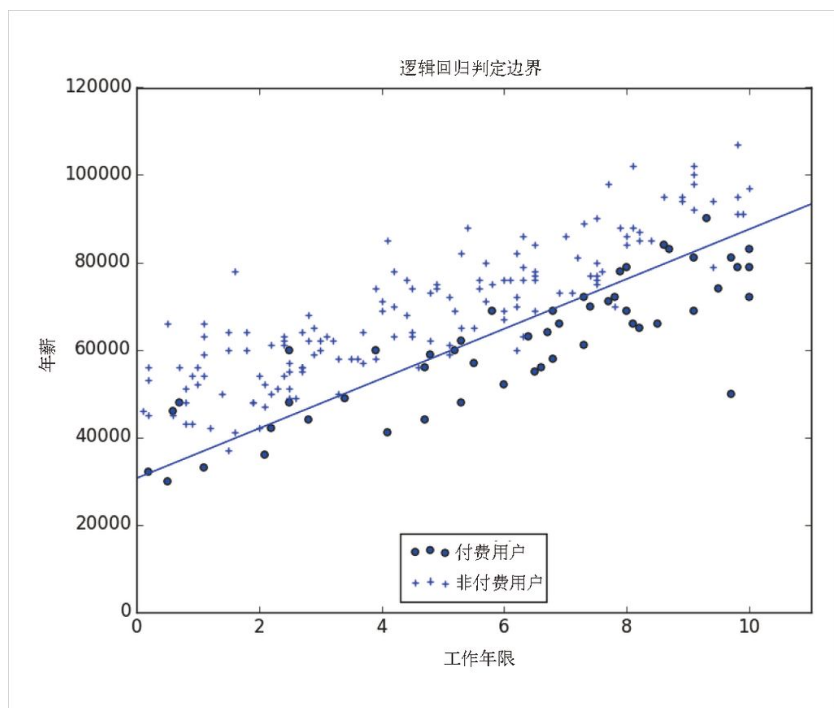


图 **16-5**：付费用户和非付费用户的判定边界

这个边界实际上就是一个超平面（hyperplane），将参数空间一分为二，一半对应着预测为付费用户，一半对应着预测为非付费用户。我们发现，这个预测只是寻找最优逻辑模型过程中的一个副产品而已。

另外，还有一种分类方法，即寻找的超平面只要对训练数据的分类效果“最佳”即可。这实际上就是支持向量机（support vector machine）思想，即寻找将距离每个类别中的最近点的距离最大化的超平面，如图 16-6 所示。

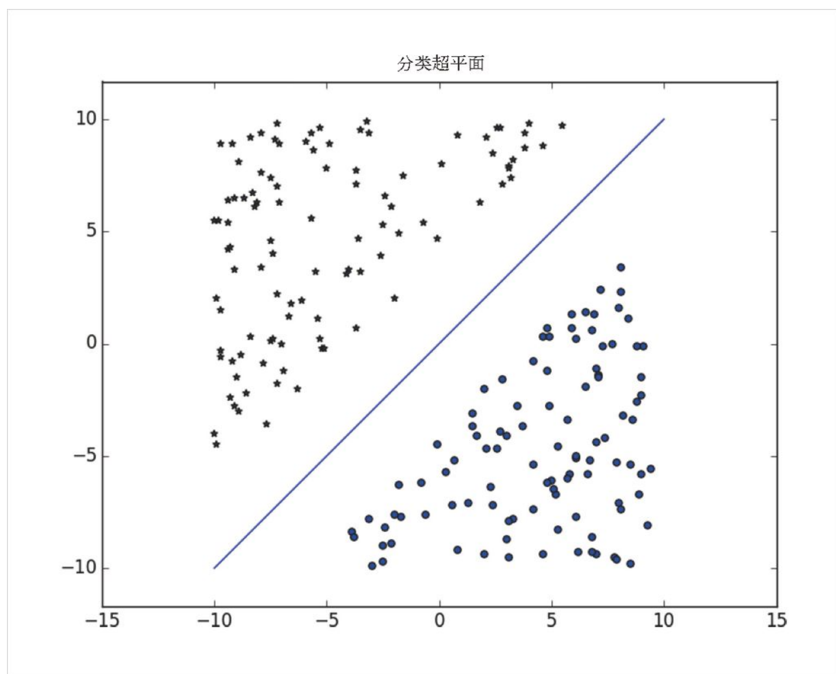


图 **16-6**：分类超平面

寻找这种超平面的过程就是一个最优化的过程，不过这里面所涉及的技术对我们来说太复杂了。另外一个不同的问题是，分类超平面也许根本就不存在。简单来说，就是在我们的“谁会付费？”的数据集中，没有能够把付费用户和非付费用户完美分隔的直线。

我们（有时）可以考虑把数据映射到一个更高维的空间中。例如，我们先看图 16-7 所示的一维数据集的情形。

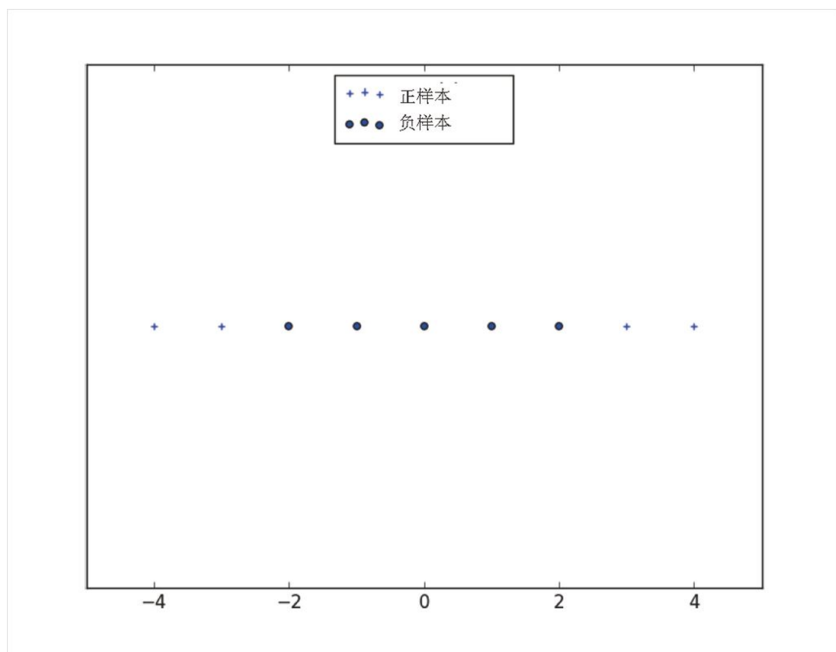


图 **16-7**：无法分隔的一维空间数据集

很明显，没有一个超平面能够将这些正样本与负样本分隔开来。但是，如果通过把 x 替换为 （x， x**2）来将数据集映射到一个二维空间，我们看一下情况会有什么变化。情况突然出现了转机：我们能够找出分隔数据的超平面了，如图 16-8 所示。

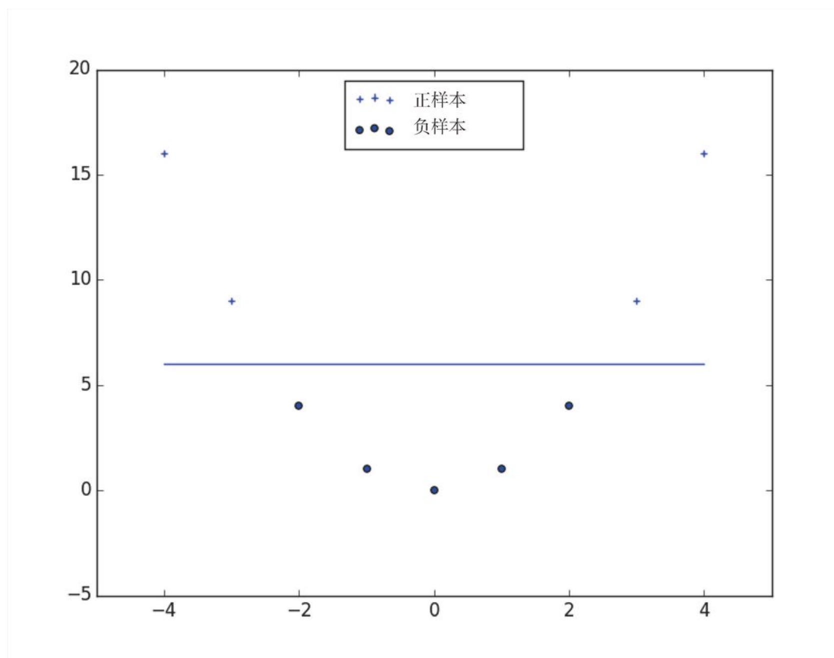


图 **16-8**：在更高维空间中数据变为可分隔的

这通常称为核方法（kernel trick），因为不用真的把数据点映射到更高维的空间（如果数据数量较多并且映射复杂的话，这个过程的代价将会很大），相反，我们可以使用“核”函数来计算更高维空间中的点积，并用它们来找出超平面。

如果不借助于专业人士编写的专门的优化软件的话，我们会很难（并且也不一定是一个好主意）利用支持向量机，因此，我们对它的介绍到此为止。

16.6　延伸学习

- scikit-learn 库同时提供了逻辑回归（http://scikit-learn.org/stable/modules/linear_model.html#logistic-regression）和支持向量机（http://scikit-learn.org/stable/modules/svm.html）的相关模块。

- scikit-learn 实际上是利用 libsvm（http://www.csie.ntu.edu.tw/~cjlin/libsvm/）来实现的支持向量机。

其网站上提供了许多支持向量机方面的优秀资料。

第 17 章　决策树

每一棵树都是一个解不开的谜。

——吉姆 · 伍德林

假设 DataSciencester 人力副总通过网站召集了许多求职者，并面试了他们，当然，对他们的满意程度各不相同。他还收集了一组数据，其中包括每个求职者的各种（定性的）特质以及面试表现情况。现在他向你咨询：能否用这些数据建立一个模型，从而识别出哪些应聘者能够通过面试？如果可以的话，那就不必浪费时间进行面试了。

这个问题看起来非常适合利用决策树（decision tree）来解决。所谓决策树，是数据科学家工具箱中的另一种预测建模工具。

17.1　什么是决策树

决策树通过树结构来表示各种可能的决策路径（decision path），以及每个路径的结果。

如果你之前玩过二十问（Twenty Questions，https://en.wikipedia.org/wiki/Twenty_Questions）这个游戏的话，那么你早就熟悉决策树了，举例来说：

- “我在猜一种动物。”
- “它有五条以上的腿吗？”
- “否。”
- “好吃吗？”
- “否。”
- “是否出现在澳大利亚五分硬币的背面？”
- “是。”
- “是针鼹吗？”

- “对，正是！”

下面是相应的判断路径。

“不超过 5 条腿”→“不好吃”→“出现在 5 分硬币上”→“针鼹！”这就是一棵特殊的（而不是非常宽泛的）“猜动物”的决策树，如图 17-1 所示。

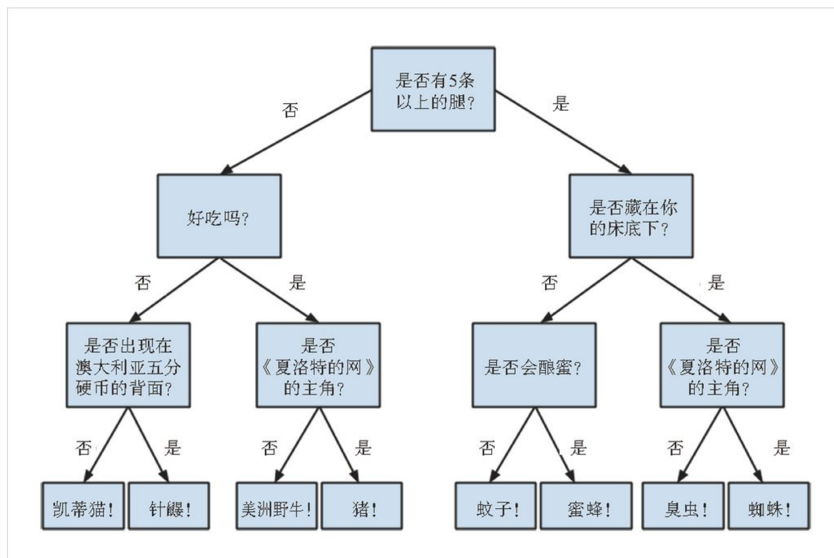


图 **17-1**：“猜动物”决策树

决策树不仅能够提供很多建议，并且非常易于理解和解释，同时，进行推断的过程还是完全透明的。与本书前面介绍的模型不同，决策树不仅可以轻松处理混合在一起的数值属性（例如腿数）和条件属性（例如好吃／不好吃），甚至还可以对缺失属性的数据进行分类。

不过，要想利用一组训练数据找出“最优”决策树却是一个非常艰巨的计算问题。（考虑到计算量的问题，我们这里要建立的是一棵足够好的决策树，而非最优决策树。即便如此，当数据集变大时，计算量仍旧会面临挑战。）更重要的是，建立决策树模型时非常容易出现对训练数据的严重过拟合现象，从而导致模型对于未曾见过的数据的效果大打折扣。关于这个问题，我们将设法加以解决。

大多数人都将决策树分为分类决策树（classification tree，它输出的是判决结果）和回归决策树（regression tree，它输出的是数值结

果）。本章我们将重点介绍分类决策树，同时还通过 ID3 算法根据已标记的数据集来确定决策树，这将有助于我们理解决策树的实际运行机制。为简单起见，我们只探讨仅有两个输出结果的问题，如“我该不该雇用这个应聘者？”“我应该给这个网站访问者展示广告 A 还是广告 B ？”或者“吃了从办公室冰箱里发现的这些食物会不会反胃？”

17.2　熵

为了建立一个决策树，我们需要决定提出哪些问题，以及这些问题的提问顺序。树的每个阶段都存在一些不确定性，其中有些不确定性已经被我们消除，而另一些则依旧存在。当知道该动物的腿不超过五条后，我们就已经排除了它是蝗虫的可能性。但是，这并没有排除它是鸭子的可能性。对于每一个可能的问题，我们都可以根据其答案对剩余的可能性空间做进一步分割。

理想情况下，我们当然愿意选择那些具有能够给决策树的预测提供更多信息的答案的问题。如果有一个是／否问题，并且答案为“是”的时候输出为 `True`，而答案为“否”的时候输出为 `False`（反之亦然），那么这样的问题自然是我们的首选了。相反，如果一个是／否问题的答案无论是啥都不能为预测提供新信息的话，那么它就不是一个好的选择。

我们用熵（entropy）这个概念来指代“信息含量”，此外，这个词还常用来表示混乱程度。在这里，我们用它来表示与数据相关的不确定性。

假设我们有一个数据集 S，每个数据元素都标明了所属的类别，即元素属于有限类别 $C_1, \cdots, C_n$ 中的一种。如果所有数据点都属于同一类别，那么也就不存在不确定性了，这就属于我们喜闻乐见的低熵情形。如果数据点均匀地分布在各个类别中，那么不确定性就较大，这时我们说具有较大的熵。

从数学的角度来讲，如果 p_i 表示 c_i 类别中的数据所占的比例，那么可以把熵定义为：

$$H(S) = -p_1 \log_2 p_1 - \cdots - p_n \log_2 p_n$$

按照通常的约定，$0\log 0 = 0$。

对于这个定义，我们不必关心其中的细枝末节，只要明白每一个

$-p_i \log_2 p_i$ 项都是非负的，并且当 p_i 接近 0 或 1 时，熵的值也接近 0（如同图 17-2 所示）即可。

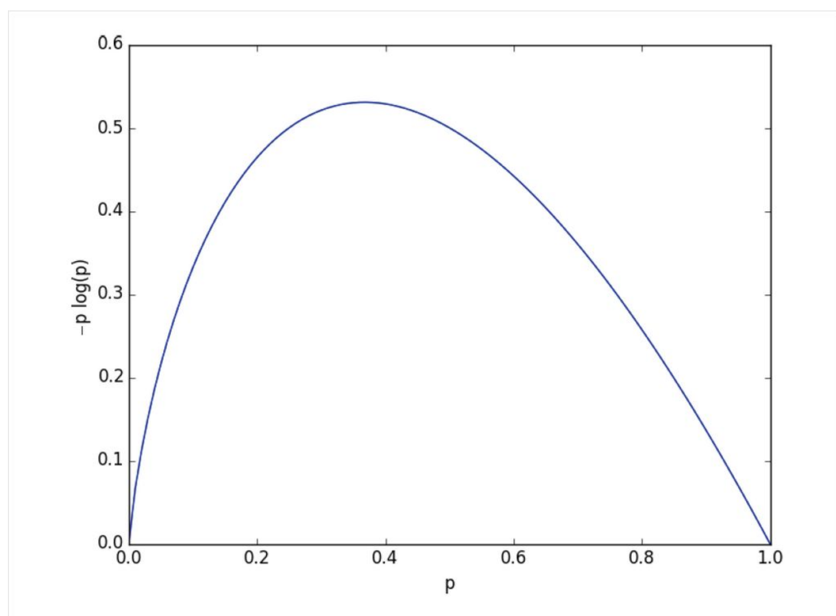


图 **17-2**：***-plog p*** 的图像

这就意味着，当每一个 p_i 越接近 0 或 1 时（即当大部分数据都属于同一个类别时），熵就越小；当许多 p_i 不接近 0 时（即当数据广泛分布于众多类别中时），熵就越大。这正是我们所期望的特性。

我们可以轻而易举地将上面的定义编写为一个函数：

```
def entropy(class_probabilities):
    """given a list of class probabilities, compute the entropy"""
    return sum(-p * math.log(p, 2)
               for p in class_probabilities
               if p)                              # 忽略零可能性
```

我们的数据点是由一对 (input, label) 组成的，这就意味着类别概率需要我们自己来计算。需要注意的是，我们并不关心标签与概率之间的关联，我们只在乎概率本身：

```
def class_probabilities(labels):
    total_count = len(labels)
```

```
    return [count / total_count
            for count in Counter(labels).values()]

def data_entropy(labeled_data):
    labels = [label for _, label in labeled_data]
    probabilities = class_probabilities(labels)
    return entropy(probabilities)
```

17.3　分割之熵

迄今为止，我们所做的是计算单组标记数据的熵（即“不确定性”）。实际上，决策树每前进一步，都要提出一个问题，而它的答案会把数据分割为一个或（更可能的情况是）多个子集。例如，“是否有五条以上的腿？”这个问题能够把动物分为五条腿以上的（如蜘蛛）和非五条腿以上的（如针鼹）。

相应地，我们希望通过某种方法对数据集的分割效果来表示熵。对于某个划分方法，如果得到的子集的熵较低（即确定性很高）的话，我们就说这个划分方法的熵较低；反之，如果得到的子集的（数量很多并且）熵较高（即不确定性很高）的话，我们就说这个划分方法的熵较高。

例如，前面“澳大利亚五分硬币”就是一个非常不明智（尽管这次非常幸运！）的问题，因为它把剩余的动物分为 S_1 ={ 针鼹 } 和 S_2={ 针鼹之外的一切动物 } 两个子集，而 S_2 这 个集合不仅过大而且高熵。（虽然 S_1 子集没有熵，但它只能代表剩余“类别”中的一小部分。）

在数学上，如果我们把数据集 S 划分为数据子集 $S_1, \cdots, S_m$，各个子集相应数据量所占比例 为 $q_1, \cdots, q_m$，那么我们就可以通过如下加权和的形式来计算这次划分的熵：

$$H = q_1 H(S_1) + \cdots + q_m H(S_m)$$

下面是具体的实现代码：

```
def partition_entropy(subsets):
    """find the entropy from this partition of data into subsets
    subsets is a list of lists of labeled data"""

    total_count = sum(len(subset) for subset in subsets)

    return sum( data_entropy(subset) * len(subset) / total_count
```

```
            for subset in subsets )
```

 这种方法的一个问题是，通过具有许多不同的值的属性进行数据划分的话，往往会由于过度拟合而导致过低的熵。例如，假设你为一家银行工作，并尝试用一些历史数据作为训练集，建立一个决策树来预测哪些客户很可能拖欠抵押贷款。我们进一步假设数据集提供了每个客户的社会保险号（SSN）。如果利用 SSN 号码对数据集进行划分，那么得到的每个子集都只含有一个人员，这样的话它们的熵必定为 0。但是，这个依赖于 SSN 号码的模型不一定适用于该训练集之外的数据。出于这个原因，你应该尽量避免使用（或直接 去掉，如果合适的话）有大量的可能取值的属性来创建决策树。

17.4　创建决策树

副总为你提供了应聘者的相关资料，包括符合你的数据规范的 `(input, label)` 对，其中每个输入都是由应聘者的各种属性构成的一个字典变量，而每个标签的取值要么为 `True`（该求职者面试成绩很好），要么为 `False`（该求职者面试成绩很差）。具体来说，数据为你提供了每个求职者的得分情况、常用语言、在 Twitter 上的活跃程度以及是否拥有博士学位等信息：

```
inputs = [
    ({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'no'},    Fals
    ({'level':'Senior', 'lang':'Java', 'tweets':'no', 'phd':'yes'},   Fals
    ({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'no'},      Tru
    ({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'no'},   Tru
    ({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'no'},       Tru
    ({'level':'Junior', 'lang':'R', 'tweets':'yes', 'phd':'yes'},     Fals
    ({'level':'Mid', 'lang':'R', 'tweets':'yes', 'phd':'yes'},         Tru
    ({'level':'Senior', 'lang':'Python', 'tweets':'no', 'phd':'no'},  Fals
    ({'level':'Senior', 'lang':'R', 'tweets':'yes', 'phd':'no'},       Tru
    ({'level':'Junior', 'lang':'Python', 'tweets':'yes', 'phd':'no'},  Tru
    ({'level':'Senior', 'lang':'Python', 'tweets':'yes', 'phd':'yes'}, Tru
    ({'level':'Mid', 'lang':'Python', 'tweets':'no', 'phd':'yes'},     Tru
    ({'level':'Mid', 'lang':'Java', 'tweets':'yes', 'phd':'no'},       Tru
    ({'level':'Junior', 'lang':'Python', 'tweets':'no', 'phd':'yes'}, Fals
]
```

我们的决策树含有许多决策节点（该节点会提出一个问题，并根据问

题的答案来指导我们下一步如何走）和叶节点（该节点为我们提供预测结果）。我们将使用较为简单的 ID3 算法来创建决策树，具体过程将在下面详细介绍。假设我们得到了一些标记过的数据，以及一个用来选择下一个分支的属性列表。

- 如果所有数据都有相同的标签，那么创建一个预测最终结果即为该标签所示的叶节点，然后停止。
- 如果属性列表是空的（即已经没有更多的问题可提问了），就创建一个预测结果为最常见的标签的叶节点，然后停止。
- 否则，尝试用每个属性对数据进行划分。
- 选择具有最低划分熵的那次划分的结果。
- 根据选定的属性添加一个决策节点。
- 针对划分得到的每个子集，利用剩余属性重复上述过程。

这就是所谓的“贪婪”算法，因为在每一步，它都会选择最快最好的那一个。对于特定的数据集，有的决策树在开头几步看起来表现不佳，但最终结果却可能是最棒的。如果是这样的话，这个算法就无法找到这样的决策树。尽管如此，它也有自己的优点，即相对来说还是很容易理解和实现的，所以，把它用作探索决策树的起点还是非常不错的。

下面让我们以手动方式在应聘者数据集上完成这些步骤。这个数据集具有 `True` 和 `False` 两种标签，我们将利用 4 个属性对其进行分类。因此，我们首先要做的就是找出熵最小的分割方法。我们将通过如下函数来完成分割：

```
def partition_by(inputs, attribute):
    """each input is a pair (attribute_dict, label).
    returns a dict : attribute_value -> inputs"""
    groups = defaultdict(list)
    for input in inputs:
        key = input[0][attribute]    # 得到特定属性的值
        groups[key].append(input)    # 然后把这个输入加到正确的列表中
    return groups
```

我们可以通过下列代码来计算熵：

```
def partition_entropy_by(inputs, attribute):
    """computes the entropy corresponding to the given partition"""
    partitions = partition_by(inputs, attribute)
    return partition_entropy(partitions.values())
```

然后我们只需要找出在整个数据集上具有最小熵的分割即可：

```
for key in ['level','lang','tweets','phd']:
    print key, partition_entropy_by(inputs, key)

# level 0.693536138896
# lang 0.860131712855
# tweets 0.788450457308
# phd 0.892158928262
```

我们看到，利用 `level` 进行的分割的熵最小，所以我们需要为每一个可能的 `level` 值建立一个子树。所有 `Mid` 应聘者都被标记成了 `True`，这意味着 `Mid` 子树是一个叶节点，其预测结果为 `True`。对于 Senior 级别的求职者，其标签既有 `True` 也有 `False`，所以我们需要进一步划分：

```
senior_inputs = [(input, label)
                 for input, label in inputs if input["level"] == "Senior"]

for key in ['lang', 'tweets', 'phd']:
    print key, partition_entropy_by(senior_inputs, key)

# lang 0.4
# tweets 0.0
# phd 0.950977500433
```

这表明，我们下一步应该根据 `tweets` 进行分割，因为它能得到熵为 0 的分割。对于 Senior 级别的应聘者，`tweets` 的值为“yes”的最终分类结果为 `True`，而 `tweets` 的值为“no”的最终分类结果为 `False`。

最后，如果我们对 Junior 级别的应聘者做同样的事情，最终会根据属性 `phd` 进行划分，并且发现没有博士学位的结果都是 `True`，拥有博士学位的结果都是 `False`。

图 17-3 为我们展示了完整的决策树。

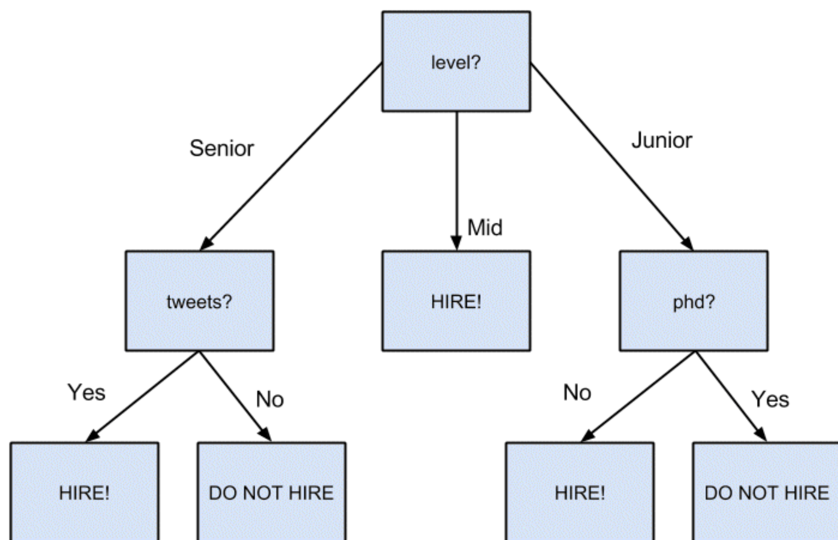


图 **17-3**：招聘决策树

17.5　综合运用

既然我们已经知道了这个算法的工作原理，下面就以更加通用的方式来实现这个算法。这意味着我们需要决定如何表示决策树。这里，我们将尽可能使用最轻量化的表示方法。我们将树定义为下列情况之一：

- `True`
- `False`
- 元组 `(attribute, subtree_dict)`

这里的 `True` 代表一个叶节点，对于任何输入该节点都会返回 `True`；`False` 也表示一个叶节点，但是对于任何输入该节点都会返回 `False`；而元组则代表一个决策节点，对于任何输入，该节点都会根据 `attribute` 的值利用相应的子树对输入进行分类。

使用这种方法，我们的招聘决策树将表示如下：

```
('level',
 {'Junior': ('phd', {'no': True, 'yes': False}),
  'Mid': True,
```

```
    'Senior': ('tweets', {'no': False, 'yes': True})})
```

不过还有一个问题需要解决，即如何处理非预期的属性值和缺失属性值的情形。如果招聘决策树遇到应聘者的 level 属性值为“Intern”的情况，该如何处置呢？我们可以通过添加一个关键字 None 来处理这种情况，这时只要把预测结果设为最常见的标签即可。（当然，如果数据集中实际上含有 None 这个值的话，这将是一个糟糕的主意。）

给定了表示方法后，我们就可以对输入进行分类了，具体如下所示：

```
def classify(tree, input):
    """classify the input using the given decision tree"""

    # 如果这是一个叶节点，则返回其值
    if tree in [True, False]:
        return tree

    # 否则这个树就包含一个需要划分的属性
    # 和一个字典，字典的键是那个属性的值
    # 值是下一步需要考虑的子树
    attribute, subtree_dict = tree

    subtree_key = input.get(attribute)     # 如果输入的是缺失的属性，则返回None

    if subtree_key not in subtree_dict:    # 如果键没有子树
        subtree_key = None                 # 则需要用到None子树

    subtree = subtree_dict[subtree_key]    # 选择恰当的子树
    return classify(subtree, input)        # 并用它来对输入分类
```

最后要做的就是利用训练数据建立决策树的具体表示形式：

```
def build_tree_id3(inputs, split_candidates=None):

    # 如果这是第一步
    # 第一次输入的所有的键就都是split candidates
    if split_candidates is None:
        split_candidates = inputs[0][0].keys()

    # 对输入里的True和False计数
    num_inputs = len(inputs)
    num_trues = len([label for item, label in inputs if label])
    num_falses = num_inputs - num_trues

    if num_trues == 0: return False     # 若没有True，则返回一个"False"叶节点
    if num_falses == 0: return True     # 若没有False，则返回一个"True"叶节点
```

```
    if not split_candidates:            # 若不再有split candidates
        return num_trues >= num_falses  # 则返回多数叶节点

    # 否则在最好的属性上进行划分
    best_attribute = min(split_candidates,
                         key=partial(partition_entropy_by, inputs))

    partitions = partition_by(inputs, best_attribute)
    new_candidates = [a for a in split_candidates
                      if a != best_attribute]

    # 递归地创建子树
    subtrees = { attribute_value : build_tree_id3(subset, new_candidates)
                 for attribute_value, subset in partitions.iteritems() }

    subtrees[None] = num_trues > num_falses      # 默认情况

    return (best_attribute, subtrees)
```

在我们所建的树上，每一个叶节点要么由清一色的 `True` 输入组成，要不就是由清一色的 `False` 输入组成。这意味着，该决策树对于这个训练数据集的预测效果堪称完美。但我们也可以把它应用到训练集之外的新数据上面：

```
tree = build_tree_id3(inputs)

classify(tree, { "level" : "Junior",
                 "lang" : "Java",
                 "tweets" : "yes",
                 "phd" : "no"} )        # True

classify(tree, { "level" : "Junior",
                 "lang" : "Java",
                 "tweets" : "yes",
                 "phd" : "yes"} )       # False
```

同时，也可以将它应用于具有缺失值或非预期值的数据：

```
classify(tree, { "level" : "Intern" } ) # True
classify(tree, { "level" : "Senior" } ) # False
```



由于我们的目的主要是演示如何构建决策树，因此这里使用了整个数据集来建立决策树。与往常一样，如果现实中我们想创造一个优秀模型的话，就应

该（收集更多的数据并且）将数据分成训练子集、验证子集和测试子集。

17.6　随机森林

由于决策树与其训练数据的契合程度非常高，因此，它总是倾向于出现过拟合现象。为了避免出现这种情况，可以使用随机森林（random forest）技术。利用该技术，我们可以建立多个决策树，然后通过投票方式决定如何对输入进行分类：

```
def forest_classify(trees, input):
    votes = [classify(tree, input) for tree in trees]
    vote_counts = Counter(votes)
    return vote_counts.most_common(1)[0][0]
```

我们知道，决策树的构建是一个确定性的过程，那么如何才能得到随机的决策树呢？

总的来说，这需要对数据进行 Bootstrap 抽样处理（这种抽样方法我们曾经在 15.6 节“题外话：Bootstrap”介绍过）。这种方法不是利用训练集合中的所有的输入数据来训练每棵决策树，而是利用 `bootstrap_sample(inputs)` 的取样结果来训练每棵决策树。因为每一棵决策树都是用不同的数据建立的，因此与其他决策树相比，每一棵都有其独特之处。（该方法的另一个好处是可以统一使用非抽样数据来测试每一棵决策树，这意味着如果你的模型效果评测方式设计得巧妙，完全可以将所有数据都用于训练集。）这种技术就是著名的 Bootstrap 集成法（bootstrap aggregating），或者简称 bagging 方法。

随机性的另一个来源是在分类时不断变换选择最佳属性（`best_attribute`）进行划分的方法。这里不是说选择全部的剩余属性进行划分，而是先从中随机选取一个子集，然后从中寻找最佳属性进行划分：

```
# 如果已经存在了几个足够的划分候选项，就查看全部
if len(split_candidates) <= self.num_split_candidates:
    sampled_split_candidates = split_candidates
# 否则选取一个随机样本
else:
    sampled_split_candidates = random.sample(split_candidates,
                                             self.num_split_candidates)
```

```
# 现在仅从这些候选项中选择最佳属性
best_attribute = min(sampled_split_candidates,
    key=partial(partition_entropy_by, inputs))

partitions = partition_by(inputs, best_attribute)
```

上面的代码展示的是一种用途更广泛的技术，称为集成学习（ensemble learning），它能够将多个较弱的模型（weak learner，通常是高偏差、低方差模型）组合成一个更加强大的模型。

随机森林是最为流行的一种集成方法，由其衍生的模型几乎到处可见。

17.7　延伸学习

- scikit-learn 库提供了许多决策树模型（http://scikit-learn.org/stable/modules/tree.html）。此外，它还提供了一个 ensemble 模块（http://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble），其中包括 `RandomForestClassifier` 以及其他集成方法。

- 我们这里仅仅介绍了决策树及其算法的皮毛知识，如果读者有志于进一步探索该主题，可以先从维基百科（http://en.wikipedia.org/wiki/Decision_tree_learning）的介绍开始学习。

第 18 章　神经网络

我喜欢胡思乱想，因为这样能唤醒脑细胞。

——苏斯博士

人工神经网络（artificial neural network，或简称神经网络）是受大脑启发而开发出来的一种预测模型。我们可以把大脑看作一团相互连接的神经元。每个神经元都以其他神经元的输出为输入，进行相应计算，如果结果超过某个阈值，则这个神经元将会进入激活状态，否则它会继续保持非激活状态。

相应地，人工神经网络则是由人工神经元组成，同样也对输入进行类似的计算。神经网络可以解决各式各样的问题，比如手写体识别与面部识别等，同时，深度学习作为数据科学最为火爆的一个分支也大量用到神经网络。然而，大部分神经网络都是些“黑盒子”，也就是说，即使考察了其工作细节，也很难获悉它们到底是如何解决问题的。此外，大型的神经网络的训练工作难度也非常大。作为一名处于“发育期”的数据科学家，你所遇到的大多数问题都不适合用神经网络来处理。有朝一日，当你试图打造一个催生奇点的人工智能的时候，或许神经网络会是个不错的选择。

18.1　感知器

感知器（perception）可能是最简单的神经网络了，或者说是由具有 n 个二进制输入的单个神经元所组成的神经网络。感知器首先会对输入值加权求和，如果加权和大于等于 0，它就会被激活：

```
def step_function(x):
    return 1 if x >= 0 else 0

def perceptron_output(weights, bias, x):
    """returns 1 if the perceptron 'fires', 0 if not"""
    calculation = dot(weights, x) + bias
    return step_function(calculation)
```

实际上，感知器只是根据点 `x` 的超平面将问题空间分隔为两部分而已：

```
```

```
dot(weights,x) + bias == 0
```

通过正确选用权值，感知器能解决许多简单的问题（图 18-1）。例如，我们可以创建一个与门（即 AND，也就是说，当两个输入都为 1 时，返回 1；只要输入有一个为 0 时，返回 0），代码如下所示：

```
weights = [2, 2]
bias = -3
```

如果两个输入都为 1，则计算结果为 2 + 2 - 3 = 1，所以输出为 1。但是，只要输入中有一个为 0，则计算结果为 2 + 0 - 3 = -1，所以输出为 0。同时，如果两个输入都为 0，则计算结果为 -3，所以输出还是 0。同样，我们还可以建立一个或门（OR），代码如下所示：

```
weights = [2, 2]
bias = -1
```

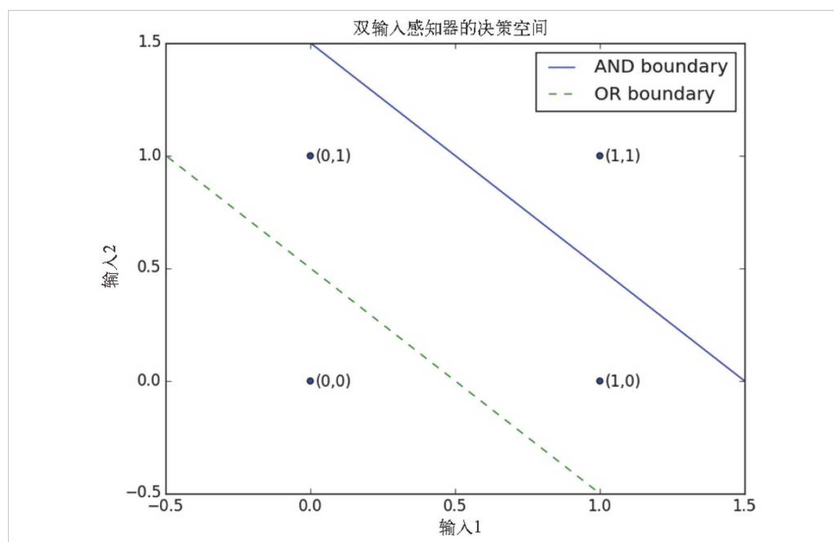


图 **18-1**：双输入感知器的决策空间

同样，我们还可以建立一个非门（即 NOT，它只有一个输入端，并且会把输入的 1 转换为 0，反之亦然），代码如下所示：

```
weights = [-2]
```

```
bias = 1
```

不过，有些问题是单个感知器所无法解决的，比如，无论你如何尝试，都无法通过一个感知器来构建异或门（XOR），即两个输入不同时输出为 1，否则输出为 0。这种情况下，我们就需要使用更加复杂的神经网络了。

当然，在建立逻辑门的时候，根本无需惟妙惟肖地模仿神经元，看一眼下面的代码你就明白了：

```
and_gate = min
or_gate = max
xor_gate = lambda x, y: 0 if x == y else 1
```

就像真实的神经元那样，当你将它们连接起来时，就会发生许多有趣的事情。

18.2　前馈神经网络

大脑的拓扑结构极为复杂，我们可以近似地把它看作一个理想化的前馈（feed-forward）神经网络，该网络由多层构成，每层由众多神经元组成，然后逐层相连。一般情况下，前馈神经网络会有一个输入层（接收输入信号，然后无需修改直接向前馈送），一个或者多个“隐藏层”（每层都是由神经元组成，这些神经元以前一层的输出作为其输入，进行某些计算，并将结果传递给下一层），以及一个输出层（这一层提供最终输出）。

正如感知器那样，每个（非输入）神经元的每个输入和偏移项都会有一个权重。为简单起见，我们将偏移项放到权重向量的末尾，并且所有神经元的偏移项的输入都是 1。

类似感知器那样，对于每个神经元而言，其输入与权重之积需要加总处理。不同之处在于，这里不是直接输出 `step_function` 函数应用于输入与权重之积的结果，而是将其平滑处理之后，输出一个近似值。准确地说，这里使用的是 `sigmoid` 函数，见图 18-2。

```
def sigmoid(t):
    return 1 / (1 + math.exp(-t))
```

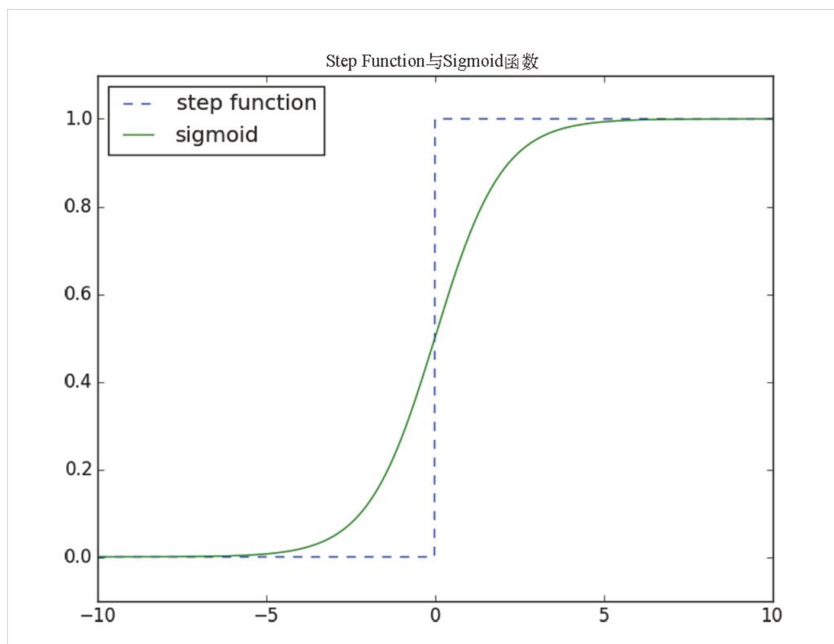


图 18-2：sigmoid 函数

为什么使用 `sigmoid` 函数，而不是更为简单的 `step_function` 函数呢？因为要训练神经网络，就得使用微积分，而要使用微积分，就得使用光滑函数。我们知道，阶梯函数无法确保处处连续，但是 `sigmoid` 函数却是它们一个非常好的平滑近似函数。



你可能还记得，我们在第 16 章也用过 `sigmoid` 函数，只不过当时我们称其为 `logistic` 函数。实际上，“sigmoid”指的是函数的外形，而“logistic”指的是这种特定的函数，但是人们经常将两者等价使用。

这样，我们就能计算其输出了，代码如下所示：

```
def neuron_output(weights, inputs):
    return sigmoid(dot(weights, inputs))
```

有了这个函数，我们就可以将神经元简单表示成一个权重列表，列表的长度等于神经元输入数量加 1，因为还要加上偏移项的权重。这样，神经网络就可以用各个（非输入）层组成的列表来表示，其中每

一层就是该层内的神经元所组成的一个列表。

也就是说，神经网络可以用（权重）列表的（神经元）列表的（层）列表来表示。

有了这种表示方法，神经网络用起来就会非常简便：

```
def feed_forward(neural_network, input_vector):
    """takes in a neural network
    (represented as a list of lists of lists of weights)
    and returns the output from forward-propagating the input"""

    outputs = []

    # 每次处理一层
    for layer in neural_network:
        input_with_bias = input_vector + [1]                  # 增加一个偏倚
        output = [neuron_output(neuron, input_with_bias)      # 计算输出
                  for neuron in layer]                        # 每一个神经元
        outputs.append(output)                                # 记住它

        # 然后下一层的输入就是这一层的输出
        input_vector = output

    return outputs
```

如今，我们无需使用感知器就能建立异或门了，这样事情就变得简单多了。所以，我们只需要调整权重，就能使得 neuron_outputs 非常接近 1 或 0 了：

```
xor_network = [# hidden layer
               [[20, 20, -30],      # 'and'神经元
                [20, 20, -10]],     # 'or'神经元
               # output layer
               [[-60, 60, -30]]]    # '第二次输入不同于第一次输入'神经元

for x in [0, 1]:
    for y in [0, 1]:
        # feed_forward生成每个神经元的输出
        # feed_forward[-1]是输出层神经元的输出
        print x, y, feed_forward(xor_network,[x, y])[-1]

# 0 0 [9.38314668300676e-14]
# 0 1 [0.9999999999999059]
# 1 0 [0.9999999999999059]
# 1 1 [9.383146683006828e-14]
```

借助于隐藏层，我们就能把一个“与”神经元和一个“或”神经元的输出馈送至“第一个输入不同于第二个输入”神经元了。这个网络所做的工作，就是判断“或运算的结果不同于与运算的结果”，这实际上就是在执行异或运算，见图 18-3。

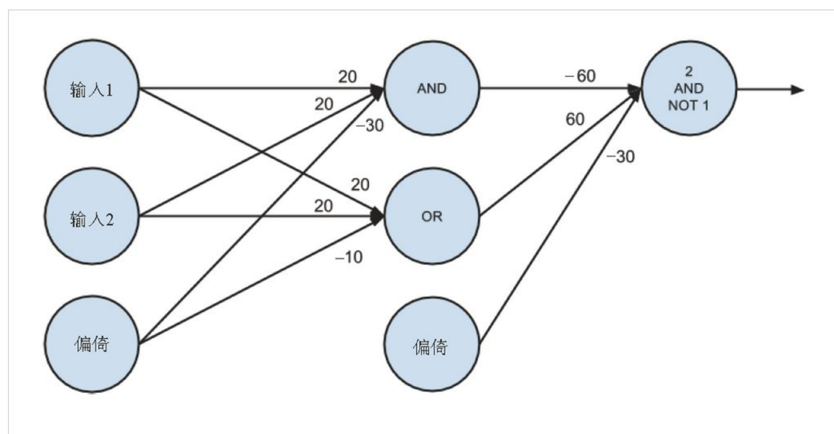


图 **18-3**：用于实现异或运算的神经网络

18.3　反向传播

通常情况下，我们是不会以手动方式建立神经网络的。部分原因在于，神经网络解决的是比较大型的问题，比如图象识别可能会用到数百或成千上万的神经元。还有一部分原因是，我们通常无法“通过推理得出”这些神经元的安排方式。

相反，我们会像往常一样使用数据用来训练神经网络。一个流行的训练算法是反向传播（backpropagation），它与前面介绍过的梯度下降法比较类似。

假如我们有一个训练集，其中含有输入向量和相应的目标输出向量。例如，前面 `xor_network` 例子中的输入向量为 `[1, 0]`，对应的目标输出端向量为 `[1]`。同时，假定我们的网络已经拥有一组权重，那么接下来，我们就需要使用以下算法来调整这些权重。

1. 在输入向量上运行 `feed_forward`，从而得到网络所有神经元的输出。

2. 这样，每个输出神经元都会得到一个误差，即目标值与输出值之

差。

3. 计算作为神经元权重的函数的误差的梯度，然后根据误差降低最快的方向调整权重。

4. 将这些输出误差反向传播给隐藏层以便计算相应误差。

5. 计算这些误差的梯度，并利用同样的方式调整隐藏层的权重。

一般情况下，这个算法需要在整个训练集上多次迭代，直到网络收敛为止：

```
def backpropagate(network, input_vector, targets):

    hidden_outputs, outputs = feed_forward(network, input_vector)

    # the output * (1 - output) is from the derivative of sigmoid
    output_deltas = [output * (1 - output) * (output - target)
                     for output, target in zip(outputs, targets)]

    # adjust weights for output layer, one neuron at a time
    for i, output_neuron in enumerate(network[-1]):
        # focus on the ith output layer neuron
        for j, hidden_output in enumerate(hidden_outputs + [1]):
            # adjust the jth weight based on both
            # this neuron's delta and its jth input
            output_neuron[j] -= output_deltas[i] * hidden_output

    # back-propagate errors to hidden layer
    hidden_deltas = [hidden_output * (1 - hidden_output) *
                      dot(output_deltas, [n[i] for n in output_layer])
                     for i, hidden_output in enumerate(hidden_outputs)]

    # adjust weights for hidden layer, one neuron at a time
    for i, hidden_neuron in enumerate(network[0]):
        for j, input in enumerate(input_vector + [1]):
            hidden_neuron[j] -= hidden_deltas[i] * input
```

实际上，以上代码所做的事情无异于显式写出与权重有关的均方误差，并应用第 8 章建立的 `minimize_stochastic` 函数。

就本例而言，显式写出梯度函数非常麻烦。如果你熟悉微积分和链式法则，那么这些数学细节对你来说还算简单，但是直接用文字表述的话（“误差函数对神经元 i 至神经元 j 输入上的权重求偏导数”）则相当无趣。

18.4　实例：战胜CAPTCHA

为了确保在网站注册的是真实的人而非“机器人”，负责产品管理的副总想在注册过程中添加 CAPTCHA 功能。

准确地讲，他想向用户展示一个数字的图片，并要求他们输入数字，以此证明他们确实是真实的人。

你告诉他这难不倒计算机，但是他却不信，所以你打算写一个可以轻松搞定这个问题的程序来说服他。

下面，我们将通过 5×5 像素的图片来显示各个数字：

```
@@@@@  ..@..  @@@@@  @@@@@  @...@  @@@@@  @@@@@  @@@@@  @@@@@  @@@@@
@...@  ..@..  ....@  ....@  @...@  @....  @....  ....@  @...@  @...@
@...@  ..@..  @@@@@  @@@@@  @@@@@  @@@@@  @@@@@  ....@  @@@@@  @@@@@
@...@  ..@..  @....  ....@  ....@  ....@  @...@  ....@  @...@  ....@
@@@@@  ..@..  @@@@@  @@@@@  ....@  @@@@@  @@@@@  ....@  @@@@@  @@@@@
```

由于我们的神经网络是以数字组成的向量作为其输入的，所以我们将每个图像转换为长度为 25 的向量，其元素的值是 1（“这个像素位于该图像中”）或 0（“这个像素不在该图像中”）。

例如，数字 0 可以表示为：

```
zero_digit = [1,1,1,1,1,
              1,0,0,0,1,
              1,0,0,0,1,
              1,0,0,0,1,
              1,1,1,1,1]
```

我们希望神经网络给出的结果能够指向一个具体的阿拉伯数字，因此我们需要 10 种不同的输出结果。例如对于数字 4 来说，正确的输出结果将是：

```
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

那么，假如我们要按顺序输入 0 到 9，则相应的识别对象为：

```
targets = [[1 if i == j else 0 for i in range(10)]
           for j in range(10)]
```

```
```

因此，四号识别对象即 `targets[4]` 的正确输出结果为数字 4。

好了，现在可以建立我们的神经网络了：

```
random.seed(0)        # 得到重复的结果
input_size = 25       # 每个输入都是一个长度为25的向量
num_hidden = 5        # 隐藏层将含有5个神经元
output_size = 10      # 对于每个输入，我们需要10个输出结果

# 每一个隐藏神经元对每个输入都有一个权重和一个偏倚权重
hidden_layer = [[random.random() for __ in range(input_size + 1)]
                for __ in range(num_hidden)]

# 每一个输出神经元对每个隐藏神经元都有一个权重和一个偏倚权重
output_layer = [[random.random() for __ in range(num_hidden + 1)]
                for __ in range(output_size)]

# 神经网络是从随机权重开始的
network = [hidden_layer, output_layer]
```

这里，我们可以通过反向传播算法来训练我们的模型：

```
# 10 000次迭代看起来足够进行收敛
for __ in range(10000):
    for input_vector, target_vector in zip(inputs, targets):
        backpropagate(network, input_vector, target_vector)
```

它在训练集上效果很好：

```
def predict(input):
    return feed_forward(network, input)[-1]

predict(inputs[7])
# [0.026, 0.0, 0.0, 0.018, 0.001, 0.0, 0.0, 0.967, 0.0, 0.0]
```

这表明，输出数字 7 的神经元的值为 0.97，而其他输出神经元的值则非常小。

同时，我们还可以将其应用于不同的数字表示形式上，比如数字 3 可以用如下表示形式：

```
predict([0,1,1,1,0,  # .@@@.
```

```
        0,0,0,1,1,  # ...@@
        0,0,1,1,0,  # ..@@.
        0,0,0,1,1,  # ...@@
        0,1,1,1,0]) # .@@@.

# [0.0, 0.0, 0.0, 0.92, 0.0, 0.0, 0.0, 0.01, 0.0, 0.12]
```

我们的神经网络认为它看起来非常像 3，但是对于像如下这种形式表示的数字 8，输出结果中数字 5、8 和 9 的得分都不低：

```
predict([0,1,1,1,0,  # .@@@.
        1,0,0,1,1,  # @..@@
        0,1,1,1,0,  # .@@@.
        1,0,0,1,1,  # @..@@
        0,1,1,1,0]) # .@@@.

# [0.0, 0.0, 0.0, 0.0, 0.0, 0.55, 0.0, 0.0, 0.93, 1.0]
```

也许更大的训练集会有所帮助。

虽然神经网络的运行不是完全透明的，但我们可以通过检查隐藏层的权重来了解它们的识别情况。特别地，我们可以把每个神经元的权重绘制为 5×5 的网格，该网格是与 5×5 的输入相对应的。

现实中，你可能希望数值为 0 的权重的颜色为白色，大于 0 的权重的绝对值越大，颜色（比如说）越绿，小于 0 的权重的绝对值越大，颜色（比如说）越红。令人遗憾的是，这在黑白色的书中是无法做到的。

相反，我们会用白色表示值为 0 的权重，其值离 0 越远的权重颜色越暗。同时，利用阴影线来表示符号为负的权重。

为此，我们需要用到函数 pyplot.imshow——这是我们之前没提及的一个函数。利用它，我们可以逐像素地绘制图像。通常情况下，这个函数对于数据科学没有很大用途，但在这里，该函数意义非凡：

```
import matplotlib
weights = network[0][0]                    # 隐藏层的第一个神经元
abs_weights = map(abs, weights)            # 阴影部分只取决于绝对值

grid = [abs_weights[row:(row+5)]           # 将权重转化为5x5的网格
        for row in range(0,25,5)]          # [weights[0:5], ..., weights[20:25]

ax = plt.gca()                             # 为了使用影线，我们需要轴
```

```
ax.imshow(grid,                              # 这里与plt.imshow一样
          cmap=matplotlib.cm.binary,         # 使用白-黑色度
          interpolation='none')              # 不进行插值处理

def patch(x, y, hatch, color):
    """return a matplotlib 'patch' object with the specified
    location, crosshatch pattern, and color"""
    return matplotlib.patches.Rectangle((x - 0.5, y - 0.5), 1, 1,
                                        hatch=hatch, fill=False, color=col

# 用交叉影线表示负权重
for i in range(5):                           # 行
    for j in range(5):                       # 列
        if weights[5*i + j] < 0:             # row i, column j = weights[5*i + j]
            # 加上黑白影线，这样无论深浅就都可见了
            ax.add_patch(patch(j, i, '/',  "white"))
            ax.add_patch(patch(j, i, '\\', "black"))

plt.show()
```

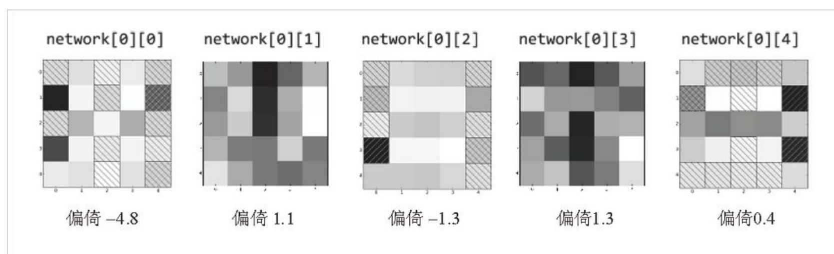


图 **18-4**：隐藏层的各个权重

通过图 18-4 可以看到，第一个隐藏神经元在左列和中间行的中心处的权重为正值，并且绝对值较大，而右列中的权重为负值，且绝对值较大。（此外，你还会发现它的偏倚项为负值，且绝对值较大，这意味着除非它“正在考察的”输入为正，否则很难被激活。）

事实上，对于这些输入来说，它的输出确实如我们所愿：

```
left_column_only = [1, 0, 0, 0, 0] * 5
print feed_forward(network, left_column_only)[0][0]   # 1.0

center_middle_row = [0, 0, 0, 0, 0] * 2 + [0, 1, 1, 1, 0] + [0, 0, 0, 0, 0
print feed_forward(network, center_middle_row)[0][0]   # 0.95

right_column_only = [0, 0, 0, 0, 1] * 5
print feed_forward(network, right_column_only)[0][0]   # 0.0
```

同样，中间的隐藏神经元似乎“喜欢”水平行而非两边的垂直行，且最后一个隐藏的神经元似乎“喜欢”中心行而非最右列。（其他两个神经元则很难解释。）

对我们以个性化形式表示的数字 3 运行这个神经网络，结果会是怎样呢？

```
my_three  = [0,1,1,1,0,   # .@@@.
             0,0,0,1,1,   # ...@@
             0,0,1,1,0,   # ..@@.
             0,0,0,1,1,   # ...@@
             0,1,1,1,0]   # .@@@.

hidden, output = feed_forward(network, my_three)
```

隐藏层的输出结果为：

```
0.121080  # 来自network[0][0]，可能是受到了(1, 4)的影响
0.999979  # 来自network[0][1]，(0, 2)和(2, 2)的贡献较大
0.999999  # 来自network[0][2]，除(3, 4)之外皆为正值
0.999992  # 来自network[0][3]，依旧是(0, 2)和(2, 2)的贡献较大
0.000000  # 来自network[0][4]，除了中间一行外，其他皆为负值或零值
```

这将进入表示“3”的输出神经元中，相应权重为 network[-1][3]：

```
-11.61  # hidden[0]的权重
 -2.17  # hidden[1]的权重
  9.31  # hidden[2]的权重
 -1.38  # hidden[3]的权重
-11.47  # hidden[4]的权重
- 1.92  # 偏倚输入的权重
```

因此，这个神经元将计算：

```
sigmoid(.121 * -11.61 + 1 * -2.17 + 1 * 9.31 - 1.38 * 1 - 0 * 11.47 - 1.92
```

正如我们所看到的，其值为 0.92。实际上，隐藏层是将 25 维空间计算成 5 个不同的分区，也就是说将每个 25 维的输入映射为 5 个数字。然后，每个输出神经元从这 5 个数字中挑出一个作为输出。

我们看到，my_three 落在分区 0（即只轻微激活了隐藏神经元

0）“下方”，前面是分区 1、2 和 3（即强烈激活那些隐藏神经元）“上部”，再往前些是分区 4 的底部（即所有神经元都没激活）。然后，10 个输出神经元中的每一个都会使用这 5 个激活单元来判断 `my_three` 是否为它们对应的数字。

18.5 延伸学习

- Coursera 提供了一门免费课程“机器学习的神经网络”（https://www.coursera.org/course/neuralnets）。在我编写本书时，最近的一次开课时间是 2012 年，不过相关课程材料仍然可用。

- Michael Nielsen 正在编写一本免费的书，书名为 *Neural Networks and Deep Learning*（http://neuralnetworksanddeeplearning.com/）。当你阅读本书时，很可能它已经写完了。

- PyBrain（http://pybrain.org/）是一个相当简单的 Python 神经网络库。

- Pylearn2（http://deeplearning.net/software/pylearn2/）是一个更加高级同时也更难使用的神经网络库。

第 19 章　聚类分析

使吾辈得以类聚者，热情而非疯狂也。

——罗伯特 · 赫里克

本书中的大多数算法都是所谓的监督学习方法，因为它们都是以一组标注过的数据作为起点的，并且在此基础上为新的、未标注过的数据做出预测。然而，本章介绍的聚类分析却是一种无监督学习方法，也就是说，它可以利用完全未经标注的数据（也可以使用标注过的数据，但我们忽略这些标签）进行工作。

19.1　原理

每当你观察某些数据源时，很可能会发现数据会以某种形式形成聚类（cluster）。例如，展示百万富翁居住地的数据集中，数据点很可能在贝弗利山和曼哈顿等地方形成聚类。而在展示人们每周工作时间（以小时为单位）的数据集中，数据则很可能聚集在 40 附近（并且，如果这些数据来自法律规定人们每周至少工作 20 个小时的国家的话，那么这些数据很可能就会聚集在 19 左右。）对于登记选民的人口统计的数据集，则可能形成多种集群（例如“足球妈妈”“无聊的退休人员”“待业千禧”等），这些群体正是民意调查和政治顾问所要密切关注的。

与之前见到的问题不同，这种问题通常没有“正确”的聚类。一个可选的聚类方案是将某些“待业千禧”与“大学毕业生”分为一伙，而将另一些“待业千禧”与“啃老族”分为一组。当然，很难说哪种方案肯定比其他方案要好，但是，对于每一种方案而言，都可以按照自己的“优良聚类”标准不断进行优化。

此外，这些聚类本身无法对自己进行标注。要想标注的话，你必须考察每个聚类中的底层数据。

19.2　模型

对于我们来说，每一个输入都是 d 维空间中的一个向量（跟以前一样，我们还是使用数字列表来表示向量）。我们的目标是识别由类似

的输入所组成的聚类，（有时）还要找出每个聚类的代表值。

例如，每个输入可以是博客文章的标题（我们可以设法用数字向量来表示它），那么在这种情况下，我们的目标可能是对相似的文章进行聚类，也可能是了解用户都在写什么博客内容。或者，假设我们有一张包含数千种（红、绿、蓝）颜色的图片，但是我们需要一个 10 色版本来进行丝网印刷。这时，聚类分析不仅可以帮助我们选出 10 种颜色，并且还能将“色差”控制在最小的范围之内。

***k*-**均值算法（*k*-means）是一种最简单的聚类分析方法，它通常需要首先选出聚类 k 的数目，然后把输入划分为集合 $S_1, \cdots, S_k$，并使得每个数据到其所在聚类的均值（中心对象）的距离的平方之和最小化。由于将 n 个点分配到 k 个聚类的方法非常多，所以寻找一个最优聚类方法是一件非常困难的事情。一般情况下，为了找到一个好的聚类方法，我们可以借助于迭代算法。

1. 首先从 d 维空间中选出选择 k 个数据点作为初始聚类的均值（即中心）。

2. 计算每个数据点到这些聚类的均值（即聚类中心）的距离，然后把各个数据点分配给离它最近的那个聚类。

3. 如果所有数据点都不再被重新分配，那么就停止并保持现有聚类。

4. 如果仍有数据点被重新分配，则重新计算均值，并返回到第 2 步。

利用第 4 章中学过的 `vector_mean` 函数，可以轻松创建如下所示的类来完成上述工作：

```
class KMeans:
    """performs k-means clustering"""

    def __init__(self, k):
        self.k = k          # 聚类的数目
        self.means = None   # 聚类的均值

    def classify(self, input):
        """return the index of the cluster closest to the input"""
        return min(range(self.k),
                   key=lambda i: squared_distance(input, self.means[i]))

    def train(self, inputs):
        # 选择k个随机点作为初始的均值
        self.means = random.sample(inputs, self.k)
        assignments = None
```

```
        while True:
            # 查找新分配
            new_assignments = map(self.classify, inputs)

            # 如果所有数据点都不再被重新分配，那么就停止
            if assignments == new_assignments:
                return

            # 否则就重新分配
            assignments = new_assignments

            # 并基于新的分配计算新的均值
            for i in range(self.k):
                # 查找分配给聚类i的所有的点
                i_points = [p for p, a in zip(inputs, assignments) if a ==

                # 确保i_points不是空的，因此除数不会是0
                if i_points:
                    self.means[i] = vector_mean(i_points)
```

下面让我们来看看其中的原理。

19.3　示例：聚会

为了庆祝 DataSciencester 的发展壮大，用户回馈部门的副总决定针对你家乡的用户组织几场私人聚会，并赞助啤酒、披萨和 DataSciencester T 恤。由于你了解所有当地用户的住址（如图 19-1），所以他想让你来选择聚会的地点，以方便大家的参与。

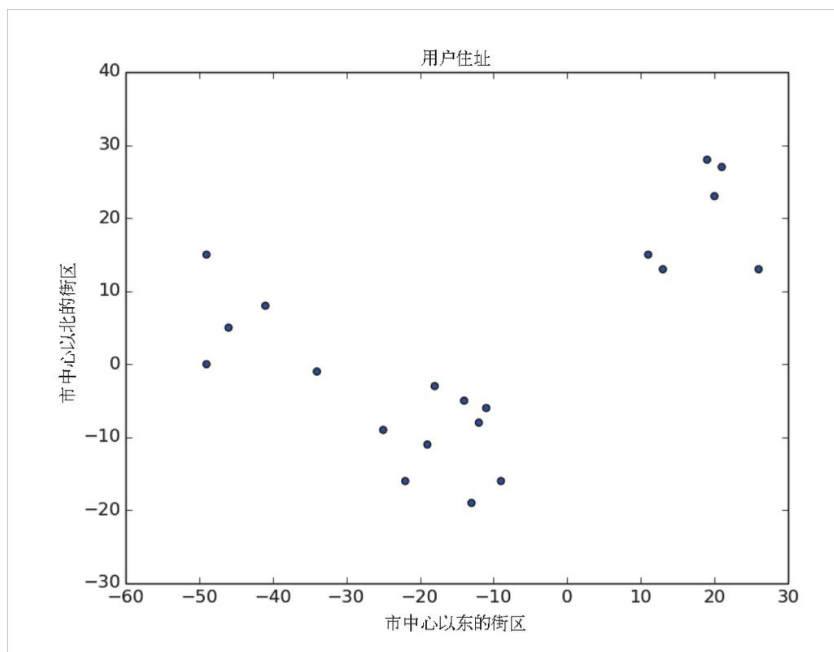


图 **19-1**：你家乡所在地的用户住址

根据具体的观察方式，你可能会发现有两个或三个用户群。（这很容易看出来，因为这里的数据只有两个维度。但是随着维度的增加，对眼神的挑战就会越来越大。）

首先，我们假设他提供的预算足以组织三次聚会。然后你来到计算机前面输入下列代码：

```
random.seed(0)            # 因此你得到的结果与我的一样
clusterer = KMeans(3)
clusterer.train(inputs)
print clusterer.means
```

你发现用户主要居住在以 [-44.5]、[-16，-10] 和 [18，20] 为中心的三个区域中，因此，你打算在这三个位置附近寻找聚会场地（见图 19-2）。

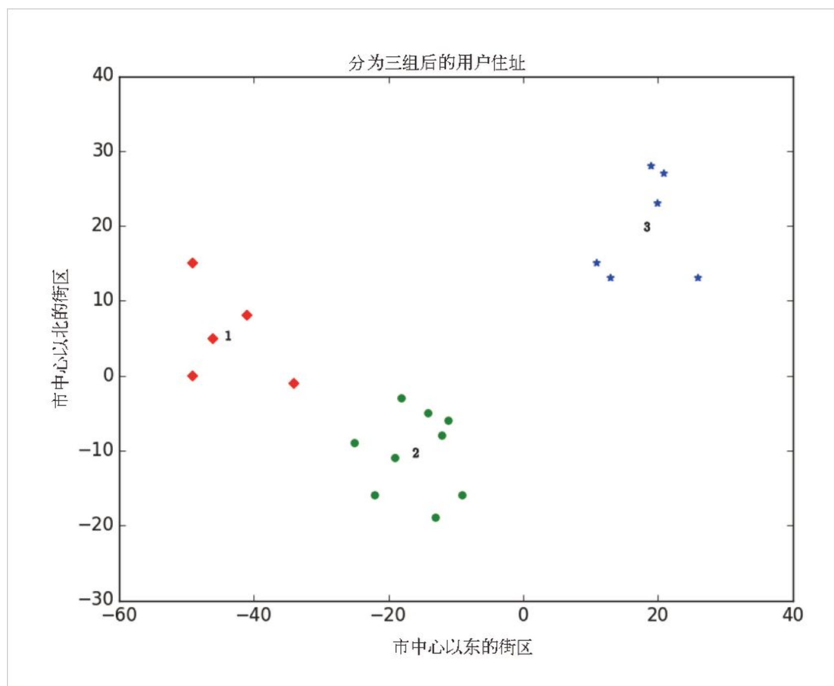


图 **19-2**：分为三组后的用户住址

你将这些汇报给了副总，但是他却告诉你目前的预算仅够组织两次聚会了。

“没问题，”你说：

```
random.seed(0)
clusterer = KMeans(2)
clusterer.train(inputs)
print clusterer.means
```

如图 19-3 所示，某次聚会地点仍然定在 [18，20] 位置附近，而另一次聚会的地点则定在 [-26，-5] 位置附近。

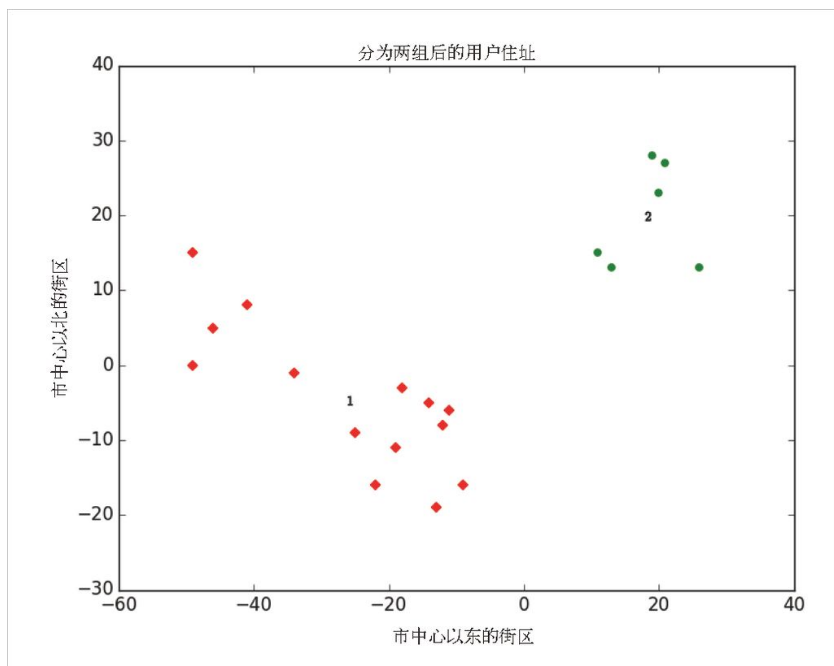


图 **19-3**：分为两组后的用户住址

19.4　选择聚类数目*k*

在前一个例子中，聚类数目 *k* 的选择是由外部因素决定的，我们无法控制。但是通常情况下，事情并非如此。*k* 的选择方法可谓五花八门，一个比较易于理解的方法是以误差（即每个数据点到所在聚类的中心的距离）的平方之和作为 *k* 的函数，画出该函数的图像，并在其“弯曲”的地方寻找合适的取值：

```
def squared_clustering_errors(inputs, k):
    """finds the total squared error from k-means clustering the inputs"""
    clusterer = KMeans(k)
    clusterer.train(inputs)
    means = clusterer.means
    assignments = map(clusterer.classify, inputs)

    return sum(squared_distance(input, means[cluster])
               for input, cluster in zip(inputs, assignments))

# 现在画出1至len（输入）的聚类图
ks = range(1, len(inputs) + 1)
errors = [squared_clustering_errors(inputs, k) for k in ks]
```

```
plt.plot(ks, errors)
plt.xticks(ks)
plt.xlabel("k")
plt.ylabel("误差的平方之和")
plt.title("总误差与聚类数目")
plt.show()
```

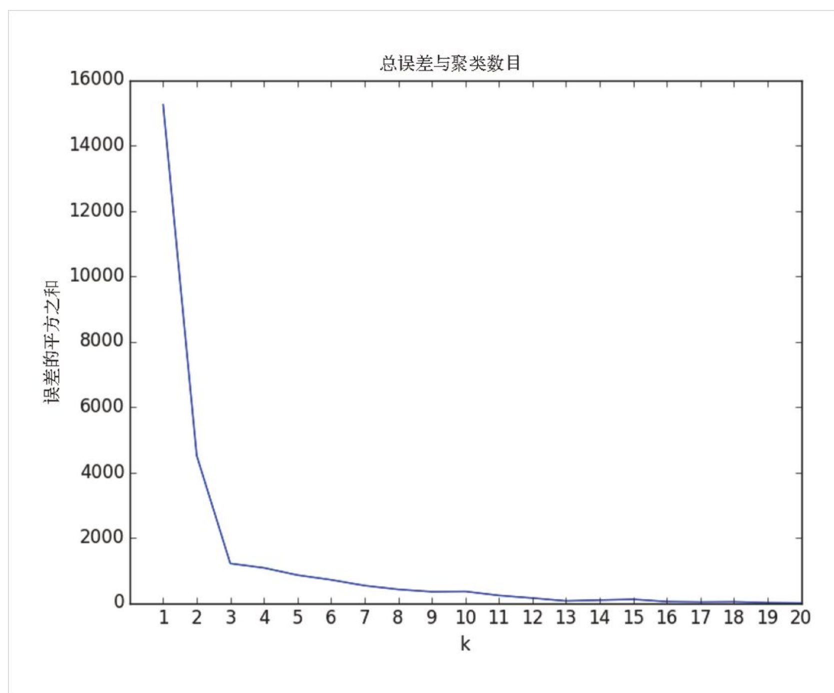


图 **19-4**：选择聚类数目 ***k***

从图 19-4 可以看出，这种方法得到的结果与我们最初的目测值是相符的，也就是说 3 是一个“合适”的聚类数目。

19.5　示例：对色彩进行聚类

负责周边产品的副总设计了一款美观的 DataSciencester 便签，希望你能够在聚会上分发给用户。令人遗憾的是，你的便签打印机功能有限，每张便签上面最多只能打出五种色彩。

同时，由于负责美术的副总正在休假，因此，负责周边产品的副总向

你咨询能否将其设计改为只包含五种颜色。

我们知道，计算机图像可以表示为像素的二维阵列，其中每个像素本身就是一个三维向量 (red, green, blue)，代表了该像素的颜色（https://en.wikipedia.org/wiki/RGB_color_model）。

为了得到图像的五色版本，我们需要执行下列步骤：

1. 选择五种颜色

2. 给每个像素从中挑选一种颜色

事实上，这个工作非常适合用 k-means 算法来做，因为该算法能够将像素划分为红 - 绿 - 蓝空间中的五个聚类。之后，我们只要将这些聚类中的像素用其中间色来重新着色就可以了。

首先，我们需要设法将图像加载到 Python 中。事实上，这可以借助 matplotlib 来实现：

```
path_to_png_file = r"C:\images\image.png"    # 不管你的图像在哪里
import matplotlib.image as mpimg
img = mpimg.imread(path_to_png_file)
```

实际上，img 在幕后是作为一个 NumPy 数组来处理的，不过就这里来说，我们可以将其视为以列表为元素的列表所组成的列表。

这里，img[i][j] 表示第 *i* 行第 *j* 列的像素，并且每个像素都由一个取值范围介于 0 和 1 之 间的 [red, green, blue] 数字列表来指定其颜色：

```
top_row = img[0]
top_left_pixel = top_row[0]
red, green, blue = top_left_pixel
```

特别是，我们可以将所有像素放到一个扁平化的列表中：

```
pixels = [pixel for row in img for pixel in row]
```

然后将其送入我们的聚类模型：



```
clusterer = KMeans(5)
clusterer.train(pixels)    # 这可能会花一些时间
```

一旦完成，我们得到了一张具有相同格式的新图像：

```
def recolor(pixel):
    cluster = clusterer.classify(pixel)           # 最近的聚类的索引
    return clusterer.means[cluster]               # 最近的聚类的均值

new_img = [[recolor(pixel) for pixel in row]      # 改变这一行像素的颜色
           for row in img] # 图像的每一行
```

接下来，我们就可以通过 plt.imshow() 来显示该图像了：

```
plt.imshow(new_img)
plt.axis('off')
plt.show()
```

当然，在只有黑白色的书中无法展示这种色彩的效果，所以我们在图 19-5 中显示了一张全彩色图片的灰度图，以及一张采用这种技术将其降低到五种颜色后的灰度图：

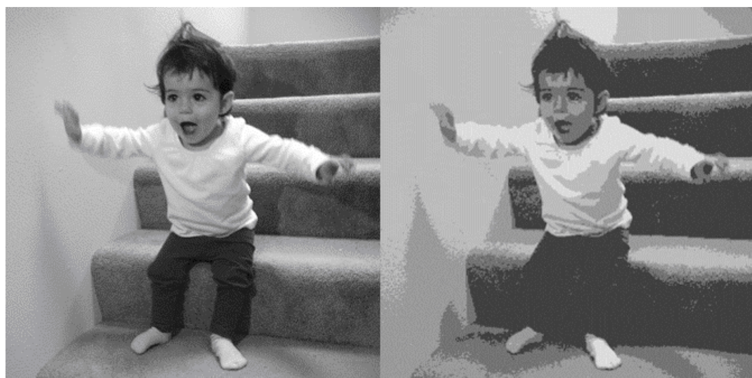


图 **19-5**：原始图像以及利用 **5-means** 去色后的效果

19.6　自下而上的分层聚类

另一种聚类方法是采用自下而上的方式“培养”聚类，为此，我们可以借助下列方式：

1. 利用每个输入构成一个聚类，当然每个聚类只包含一个元素；

2. 只要还剩余多个聚类，就找出最接近的两个，并将它们合二为一。

最后，我们将得到一个包含所有输入的巨大的聚类。如果我们将合并顺序记录下来，就可以通过拆分的方法来重建任意数量的聚类。举例来说，如果我们想得到 3 个聚类，那么只要撤销最后两次合并就可以了。

我们将使用一种非常简单的方法来表示聚类。首先，我们的数值将进入叶（leaf）聚类中，这时我们将其表示为一元组：

```
leaf1 = ([10, 20],)    # 要创建一元组，需要在末尾加一个逗号
leaf2 = ([30, -15],)   # 否则Python会把括号当成单纯的括号
```

我们通过合并上面的聚类来培育新的聚类，并将其记为二元组（合并次序，子聚类）：

```
merged = (1, [leaf1, leaf2])
```

我们稍后会介绍合并次序，但现在不妨先来创建一些辅助函数：

```
def is_leaf(cluster):
    """a cluster is a leaf if it has length 1"""
    return len(cluster) == 1

def get_children(cluster):
    """returns the two children of this cluster if it's a merged cluster;
    raises an exception if this is a leaf cluster"""
    if is_leaf(cluster):
        raise TypeError("a leaf cluster has no children")
    else:
        return cluster[1]

def get_values(cluster):
    """returns the value in this cluster (if it's a leaf cluster)
    or all the values in the leaf clusters below it (if it's not)"""
    if is_leaf(cluster):
        return cluster      # 已经是一个包含值的一元组
    else:
        return [value
                for child in get_children(cluster)
                for value in get_values(child)]
```

为了合并相距最近的聚类，我们需要明确聚类之间的距离的概念。为此，我们将使用两个聚类的元素之间的最小距离，据此将两个挨得最近的聚类合并（但有时会产生巨大的链式聚类，但是聚类之间却挨得不是很紧）。如果想得到紧凑的球状聚类，可使用最大距离，而不是最小距离，因为使用最大距离合并聚类时，它会尽力将两者塞进一个最小的球中。实际上，这两种距离都很常用，就像平均距离也很常用一样：

```
def cluster_distance(cluster1, cluster2, distance_agg=min):
    """compute all the pairwise distances between cluster1 and cluster2
    and apply _distance_agg_ to the resulting list"""
    return distance_agg([distance(input1, input2)
                         for input1 in get_values(cluster1)
                         for input2 in get_values(cluster2)])
```

我们将借助合并次序踪迹（slot）来跟踪合并的顺序。这个数字越小，表示合并的次序越靠后。这意味着，当我们想分拆聚类的时候，可以根据合并次序的值，从最小到最大依次进行。由于叶聚类不是合并而来的（这意味着无需分拆它们），因此，我们将它们合并次序的值规定为无穷大：

```
def get_merge_order(cluster):
    if is_leaf(cluster):
        return float('inf')
    else:
        return cluster[0]  # merge_order是二元组中的第一个元素
```

现在我们可以创建聚类算法了：

```
def bottom_up_cluster(inputs, distance_agg=min):
    # 最开始每个输入都是一个叶聚类/一元组
    clusters = [(input,) for input in inputs]

    # 只要剩余一个以上的聚类……
    while len(clusters) > 1:
        # 就找出最近的两个聚类
        c1, c2 = min([(cluster1, cluster2)
                      for i, cluster1 in enumerate(clusters)
                      for cluster2 in clusters[:i]],
                      key=lambda (x, y): cluster_distance(x, y, distance_a

        # 从聚类列表中将它们移除
        clusters = [c for c in clusters if c != c1 and c != c2]

        # 使用merge_order = 剩余聚类的数目来合并它们
```

```
        merged_cluster = (len(clusters), [c1, c2])

        # 并添加它们的合并
        clusters.append(merged_cluster)

    # 当只剩一个聚类时，返回它
    return clusters[0]
```

它的使用方法非常简单：

```
base_cluster = bottom_up_cluster(inputs)
```

这将得到一个聚类，简单表示如下：

```
(0, [(1, [(3, [(14, [(18, [([19, 28],),
                           ([21, 27],)]),
                     ([20, 23],)]),
               ([26, 13],)]),
          (16, [([11, 15],),
                 ([13, 13],)])]),
    (2, [(4, [(5, [(9, [(11, [([-49, 0],),
                               ([-46, 5],)]),
                         ([-41, 8],)]),
                   ([-49, 15],)]),
              ([-34, -1],)]),
         (6, [(7, [(8, [(10, [([-22, -16],),
                               ([-19, -11],)]),
                         ([-25, -9],)]),
                   (13, [(15, [(17, [([-11, -6],),
                                      ([-12, -8],)]),
                                ([-14, -5],)]),
                          ([-18, -3],)])]),
              (12, [([-13, -19],),
                     ([-9, -16],)])])])])
```

对于每一个合并而来的聚类，我都会将其子聚类纵向连接。当我们说“0 号聚类”为合并次序为 0 的聚类时候，你可以将此理解为：

- 0 号聚类是由 1 号聚类和 2 号聚类合并得到的；
- 1 号聚类是由 3 号聚类和 16 号聚类合并得到的；
- 16 号聚类是由叶节点 `[11, 15]` 和叶节点 `[13, 13]` 合并得到的。

- 以此类推……

因为我们有 20 个输入，所以只要经过 19 次合并，便能得到这个聚类。第一个合并而来的聚类是通过合并叶节点 `[19, 28]` 和叶节点 `[21, 27]` 得到的。而最后一个合并而来的聚类则是 0 号聚类。

但是，一般情况下我们不喜欢这种繁琐的文字表达方法。（不过话又说回来，对于创建一个用户友好型的可视化聚类分层结构，这也算是一个有趣的练习。）下面让我们来写一个函数，使其可以通过执行适当次数的分拆动作来产生任意数量的聚类：

```
def generate_clusters(base_cluster, num_clusters):
    # 开始的列表只有基本聚类
    clusters = [base_cluster]

    # 只要我们还没有足够的聚类……
    while len(clusters) < num_clusters:
        # 选择上一个合并的聚类
        next_cluster = min(clusters, key=get_merge_order)
        # 将它从列表中移除
        clusters = [c for c in clusters if c != next_cluster]
        # 并将它的子聚累添加到列表中（即拆分它）
        clusters.extend(get_children(next_cluster))

    # 一旦我们有了足够的聚类……
    return clusters
```

举例来说，如果我们想要生成三个聚类，可以使用下列代码：

```
three_clusters = [get_values(cluster)
                  for cluster in generate_clusters(base_cluster, 3)]
```

利用下面的代码，我们可以轻松绘制其图形：

```
for i, cluster, marker, color in zip([1, 2, 3],
                                     three_clusters,
                                     ['D','o','*'],
                                     ['r','g','b']):
    xs, ys = zip(*cluster)  # 魔法般的解压方式
    plt.scatter(xs, ys, color=color, marker=marker)

    # 向聚类的均值添加一个数字
    x, y = vector_mean(cluster)
    plt.plot(x, y, marker='$' + str(i) + '$', color='black')

plt.title("利用最短距离得到的三个自下而上的聚类")
```

```
plt.xlabel("市中心以东的街区")
plt.ylabel("市中心以北的街区")
plt.show()
```

与 *k*- 均值算法相比，我们得到了一个大不相同的结果，具体如图 19-6 所示。

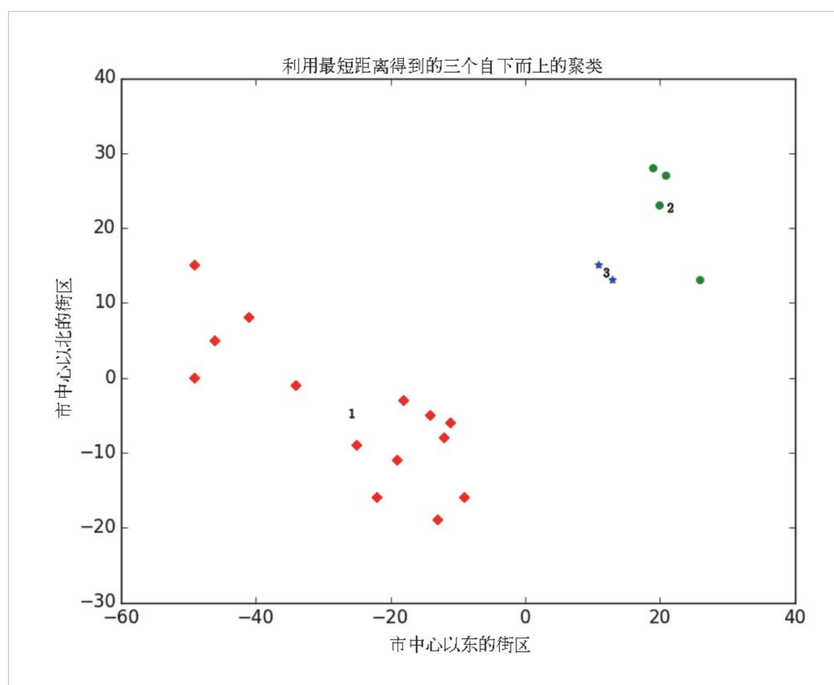


图 **19-6**：利用最短距离得到的三个自下而上的聚类

正如我们上面提到的，这是因为在 `cluster_distance` 函数中使用参数 `min` 时往往会得到链状聚类。相反，如果我们使用参数 `max`（这能得到更加紧凑的聚类），将得到看上去与图 19-7 所示的 3-means 无异的结果。

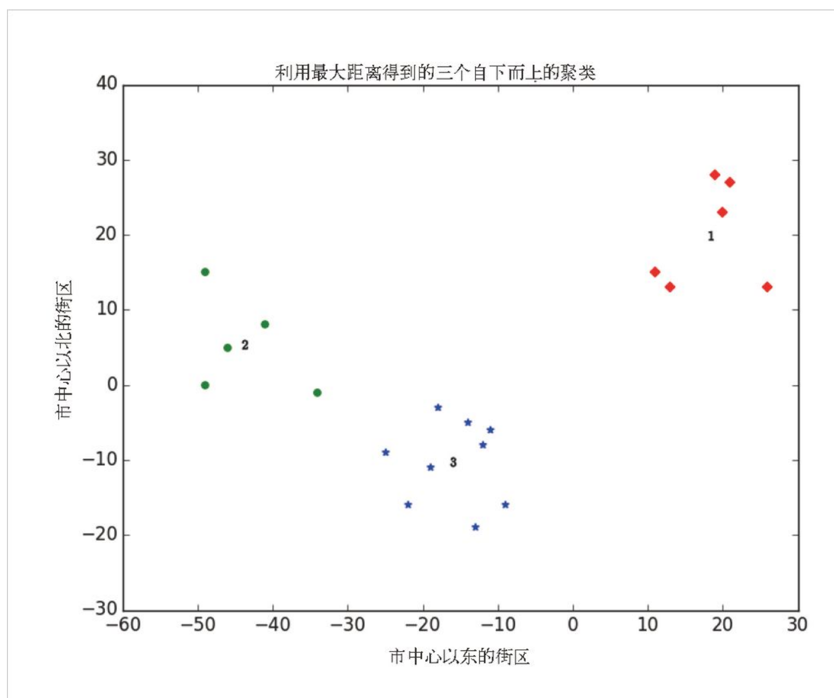


图 19-7：利用最大距离得到的三个自下而上的聚类

 以上的 `bottom_up_clustering` 的实现代码相对来说已经很简单了，但是计算效率依然低得吓人。特别是，在每一步它都要重新计算每对输入之间的距离。更有效的实现方法是，预先算出每对输入之间的距离，然后在 cluster_distance 里面进行查找。一个真正高效的实现方法可能还需要存储上一步的 `cluster_distance`。

19.7　延伸学习

- `scikit-learn` 库中提供了一个单独的模块 `sklearn.cluster`（http://scikit-learn.org/stable/modules/clustering.html），其中含有多个聚类算法，包括 `KMeans` 和 `Ward` 分级聚类算法（该算法使用了不同的聚类合并规则）。

- Scipy（http://www.scipy.org/）模块也有两个聚类模型，即 `scipy.cluster.vq`（它使用 *k*-均值算法）模型和

`scipy.cluster.hierarchy`（它使用多种层次聚类算法）模型。

第 20 章　自然语言处理

他们刚从一场语言的盛宴上偷了些残羹冷炙回来。

——威廉 · 莎士比亚

自然语言处理（natural language processing，NLP）是指与语言有关的各种计算技术。这是一个广泛的领域，但我们这里只介绍几种相关的技术，它们简约却不简单。

20.1　词云

在第 1 章中，我们曾经计算过用户兴趣词汇的数量。为了使单词及其数量可视化，一种方法是使用词云，它不仅能够以艺术化的形式展示单词，而且还能使单词的大小与其数量呈正比。

但是，一般情况下，数据科学家并不看重词云，这在很大程度上是因为单词的布局没有任何特殊意义，顶多意味着“这里还有一些空，可以放上一个单词”而已。

当你不得不生成一个词云的时候，不妨考虑一下能否透过词的坐标传达某些东西。举例来说，假如你收集了一些与数据科学相关的流行语，那么对于每一个流行语，你可以用两个介于 0 至 100 之间的数字来描述，第一个数字代表它在招聘广告中出现的频次，第二个数字是在简历中出现的频次：

```
data = [ ("big data", 100, 15), ("Hadoop", 95, 25), ("Python", 75, 50),
         ("R", 50, 40), ("machine learning", 80, 20), ("statistics", 20, 6
         ("data science", 60, 70), ("analytics", 90, 3),
         ("team player", 85, 85), ("dynamic", 2, 90), ("synergies", 70, 0)
         ("actionable insights", 40, 30), ("think out of the box", 45, 10)
         ("self-starter", 30, 50), ("customer focus", 65, 15),
         ("thought leadership", 35, 35)]
```

词云的做法，只不过就是利用很酷的字体把各个单词布置到页面上罢了（图 20-1）。



图 **20-1**：由热门术语组成的词云

这看起来虽然整洁，但并没有告诉我们任何事情。一个更有趣的方法可能是将它们分散开来，利用水平位置表示其在招聘广告中的流行度，用垂直位置表示其在简历中的流行度，这样就能形象地传达一些信息（图 20-2）：

```
def text_size(total):
    """equals 8 if total is 0, 28 if total is 200"""
    return 8 + total / 200 * 20

for word, job_popularity, resume_popularity in data:
    plt.text(job_popularity, resume_popularity, word,
             ha='center', va='center',
             size=text_size(job_popularity + resume_popularity))
plt.xlabel("Popularity on Job Postings")
plt.ylabel("Popularity on Resumes")
plt.axis([0, 100, 0, 100])
plt.xticks([])
plt.yticks([])
plt.show()
```

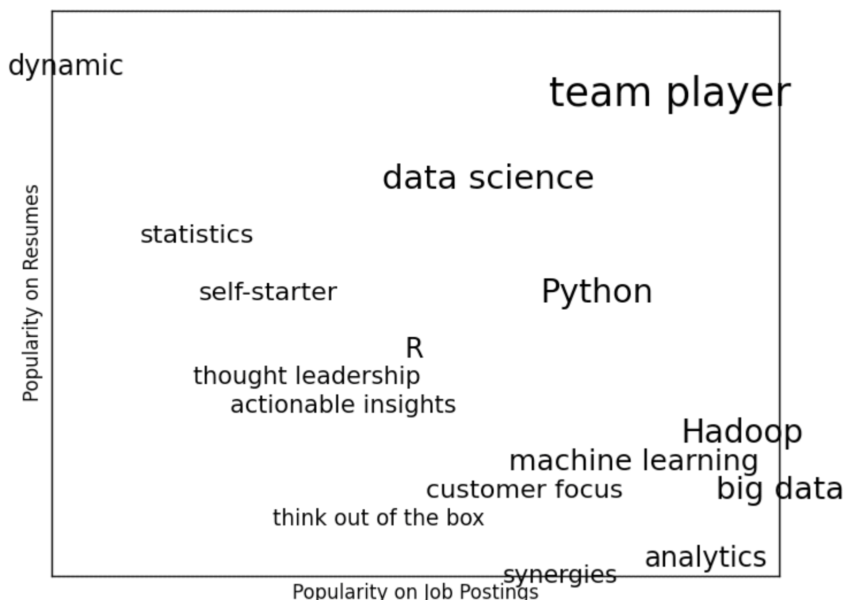


图 **20-2**：一个更有意义（尽管不如先前那么美观）的词云

20.2　n-grams模型

DataSciencester 负责搜索引擎营销的副总突发奇想，要创建数以千计的数据科学方面的 Web 页面，以便人们在搜索与数据科学有关的词语时，我们的网站能够在搜索结果中的排名更加靠前。（你试图向他解释，由于搜索引擎的算法已经足够聪明了，所以这种做法很难奏效，但是他根本就不听这一套。）

当然，他既不想亲自编写数以千计的网页，也不打算雇用一批“水军”来做这件事情。相反，他向你咨询是否可以通过编程方式来生成这些网页。为此，我们需要寻找某种方法来对语言进行建模。

这种方法当然是有的，比如首先搜集一批文档，然后利用统计方法得到一个语言模型。在我们的例子中，我们将从 Mike Loukides 的文章“什么是数据科学？”（https://www.oreilly.com/ideas/what-is-data-science）着手。

就像在第 9 章中所做的那样，我们将使用一些 Web 请求命令（`requests`）和 `BeautifulSoup` 来检索数据。不过，这里有几个问题需要引起我们的注意。

第一个问题是，文本中的单引号实际上就是 Unicode 字符 u"\u2019"。我们可以创建一个辅助函数，用正常的单引号来取代它们：

```
def fix_unicode(text):
    return text.replace(u"\u2019", "'")
```

第二个问题是，在获得了网页的文本之后，我们需要把它做成一个由单词和句号组成的序列（这样我们就可以知道句子的结尾在哪里）。为此，我们可以借助于 re.findall() 函数来完成：

```
from bs4 import BeautifulSoup
import requests
url = "http://radar.oreilly.com/2010/06/what-is-data-science.html"
html = requests.get(url).text
soup = BeautifulSoup(html, 'html5lib')

content = soup.find("div", "entry-content")   # 找到entry-content div
regex = r"[\w']+|[\.]"                        # 匹配一个单词或一个句点

document = []

for paragraph in content("p"):
    words = re.findall(regex, fix_unicode(paragraph.text))
    document.extend(words)
```

当然，我们可以（也应该）进一步清理这些数据。文档中依然存在一些多余的文字（例如，第一个字“Section”就多余），同时，我们是利用句点来断句的（例如，遇到“Web 2.0”就会出问题）。此外，文档中还散布了一些标题和列表。尽管如此，这个文档已经可以凑合着用了。

将文本做成了单词序列之后，我们就可以通过以下方式对语言进行建模了：给定某个起始单词（比如“book”），我们可以找出源文档中所有在它后面出现过的那些单词（这里是“isn't”“a”“shows”“demonstrates”和“teaches”）。我们从中随机选择一个来作为下一个单词，然后重复这个过程，直到我们遇到一个句点为止，因为句点就意味着句子的结束。我们将这个模型称之为二元模型（bigram model），因为这完全是由原始数据中 2 个词（一个词对）同时出现的频率决定的。

那么起始单词呢？实际上，我们只要从句点后面的单词中随机选择就

行了。首先，让我们预先计算出可能的单词语次转变。回想一下，对于 zip 来说，只要输入中有一个已经处理完毕，它就会停下来，因此，我们可以利用 `zip(document, document[1:])` 求出文档中有多少对连续元素：

```
bigrams = zip(document, document[1:])
transitions = defaultdict(list)
for prev, current in bigrams:
    transitions[prev].append(current)
```

下面我们就可以生成句子了：

```
def generate_using_bigrams():
    current = "."   # 这意味着下一个单词是一个新句子的开头
    result = []
    while True:
        next_word_candidates = transitions[current]    # 双连词 (current, _
        current = random.choice(next_word_candidates)  # 随机选择一个
        result.append(current)                         # 将其附加到结果中
        if current == ".": return " ".join(result)     # 如果是"."，就完成了
```

它产生的那些句子都是些无意义的数据，不过你可以把这些句子放到网站上，以让网站看起来更能与数据科学挂钩。举例来说：

> If you may know which are you want to data sort the data feeds web friend someone on trending topics as the data in Hadoop is the data science requires a book demonstrates why visualizations are but we do massive correlations across many commercial disk drives in Python language and creates more tractable form making connections then use and uses it to solve a data.
>
> ——Bigram Model

如果我们使用三元模型（trigrams）的话，就能够降低这些句子无意义的程度。所谓三元模型，就是使用三个连续的词得到的模型。（更一般地讲，你还可以考虑由 *n* 个连续的单词得到的 *n*-grams 模型，不过对于我们来说，由三个词组成的就足够了。）现在，这种语次转变将取决于前两个单词：

```
trigrams = zip(document, document[1:], document[2:])
```

```
trigram_transitions = defaultdict(list)
starts = []

for prev, current, next in trigrams:

    if prev == ".":                  # 如果前一个"单词"是个句点
        starts.append(current)       # 那么这就是一个起始单词

    trigram_transitions[(prev, current)].append(next)
```

需要注意的是，现在我们必须将这些起始词单独记录下来。我们可以使用几乎相同的方法来生成句子：

```
def generate_using_trigrams():
    current = random.choice(starts)    # 随机选择一个起始单词
    prev = "."                               # 前面加一个句点'.'
    result = [current]
    while True:
        next_word_candidates = trigram_transitions[(prev, current)]
        next_word = random.choice(next_word_candidates)
        prev, current = current, next_word
        result.append(current)

        if current == ".":
            return " ".join(result)
```

这次得到的句子看起来要好一些：

> In hindsight MapReduce seems like an epidemic and if so does that give us new insights into how economies work That's not a question we could even have asked a few years there has been instrumented.
>
> ——Trigram Model

当然，它们之所以看起来更好一些，是因为生成过程的每一步中，所面临的选择要更少一些，甚至有时候只有一种选择。这就意味着生成的句子（或至少是长短语）经常跟原始数据中的一字不差。更多的数据会有所帮助；此外，如果从多篇数据科学方面的文章中收集 *n*-grams，收到的成效会更好。

20.3　语法

还有一种语言建模方法，那就是利用语法规则（grammar）来生成符合要求的句子。在小学的时候，我们就已经知道了词的词类及其组合方式。例如，如果你有一个非常糟糕的英语老师，你必定会认为句子都是由名词后面跟动词构成的。这样的话，如果给你一个由名词和动词组成的列表，你就可以根据这种规则来造句了。

下面，我们将定义一个稍微复杂的语法：

```
grammar = {
    "_S"  : ["_NP _VP"],
    "_NP" : ["_N",
             "_A _NP _P _A _N"],
    "_VP" : ["_V",
             "_V _NP"],
    "_N"  : ["data science", "Python", "regression"],
    "_A"  : ["big", "linear", "logistic"],
    "_P"  : ["about", "near"],
    "_V"  : ["learns", "trains", "tests", "is"]
}
```

我们约定，以下划线开头的名称表示语法规则，它们需要进一步展开；而其他名称是不需要进一步处理的终端符号。

例如，"_S" 是“句子”规则，其产生一个 "_NP"（“名词短语”）规则，后面紧跟一个 "_VP"（“动词短语”）规则。

动词短语规则可能会产生一个 "_V"（“动词”）规则，也可能会产生一个动词规则继之以名词短语规则。

请注意，"_NP" 规则所生成的规则中也包括其自身。我们知道，语法是可以递归的，因此，尽管这里这些语法非常有限，但是照样能够产生无穷多不同的句子。

那么，我们如何通过这些语法来生成句子呢？我们不妨从一个包含句子规则的列表 ["_S"] 着手。然后，我们将不断展开每一项规则，即从该规则的产物中随机选择一个来代替它。当我们的列表元素全部变成终端符号时，我们就可以停下来了。

举例来说，上述过程可能像下面这样：

```
['_S']
['_NP','_VP']
['_N','_VP']
['Python','_VP']
```

```
['Python','_V','_NP']
['Python','trains','_NP']
['Python','trains','_A','_NP','_P','_A','_N']
['Python','trains','logistic','_NP','_P','_A','_N']
['Python','trains','logistic','_N','_P','_A','_N']
['Python','trains','logistic','data science','_P','_A','_N']
['Python','trains','logistic','data science','about','_A', '_N']
['Python','trains','logistic','data science','about','logistic','_N']
['Python','trains','logistic','data science','about','logistic','Python']
```

我们如何实现它呢？ 首先，我们需要创建一个简单的辅助函数来识别终端符号：

```
def is_terminal(token):
    return token[0] != "_"
```

接下来，我们需要编写一个函数，将一个标记列表变成一个句子。首先，我们需要找到第一个非终结符号标记。如果我们找不到这种标记，那就意味着我们已经有一个完整的句子，可以收工了。

如果我们真的找到了一个非终端符号，那么就在其产物中随机选择一个。如果选中的是个终端符号（即一个单词），那么直接用它替换相应的标记即可。除此之外，如果选中的是一个由空格符分隔的非终端符标记，那么则需要进行拆分，并将其拼接到当前标记中。总之，我们的工作就是在一组新标记上不断重复这个过程。

上述过程可以通过下列代码实现：

```
def expand(grammar, tokens):
    for i, token in enumerate(tokens):

        # 跳过终端符号
        if is_terminal(token): continue

        # 如果这一步我们发现了一个非终端符号
        # 需要随机选择一个替代者
        replacement = random.choice(grammar[token])

        if is_terminal(replacement):
            tokens[i] = replacement
        else:
            tokens = tokens[:i] + replacement.split() + tokens[(i+1):]

        # 现在展开新的符号列表
        return expand(grammar, tokens)
```

```
    # 如果到达这一步，就找出了所有的终端符号，可以收工了
    return tokens
```

现在我们可以生成句子了：

```
def generate_sentence(grammar):
    return expand(grammar, ["_S"])
```

只要我们不断改变语法——例如添加更多的单词、添加更多的规则、添加各种词类等——就能得到足够多的网页来满足公司的需要。

实际上，当语法用于另一个方向时，会变得更加有趣。给定一个句子，我们就可以用语法来解析句子。这就能帮助我们识别主语和动词，从而理解句子的含义。

用数据科学来生成文本的确是个妙招，但更神奇的是，它还可以用来理解文本。（这方面的程序库请参阅 16.6 节“延伸学习”部分。）

20.4　题外话：吉布斯采样

根据一些概率分布来生成样本是非常简单的事情。我们可以通过以下这行代码得到一些均匀分布的随机变量：

```
random.random()
```

以及用以下代码得到一些正态分布的随机变量：

```
inverse_normal_cdf(random.random())
```

但某些概率分布却很难进行采样。当我们只知道一些条件分布时，可以通过吉布斯采样技术根据多维分布来生成样本。

例如，假设我们在掷两只骰子。这里用 x 表示第一骰子的点数，y 表示两个骰子的点数之和。假设我们要产生大量形如 (x, y) 的数据对，这种情况下，我们可以直接通过下列代码来轻松生成所需样本：

```
def roll_a_die():
    return random.choice([1,2,3,4,5,6])
```

```
def direct_sample():
    d1 = roll_a_die()
    d2 = roll_a_die()
    return d1, d1 + d2
```

但是，这里假设你只知道条件分布。在已知道 x 的条件下求 y 的分布是很容易的：如果你知道了 x 的值，那么 y 就有同等机会等于 $x+1$，$x+2$，$x+3$，$x+4$，$x+5$，$x+6$：

```
def random_y_given_x(x):
    """equally likely to be x + 1, x + 2, ... , x + 6"""
    return x + roll_a_die()
```

如果将已知条件反过来，事情会变得更加复杂。举例来说，如果你知道 y 为 2，那么 x 必定为 1（因为只有当两个骰子的点数都为 1 的时候，点数之和才可能为 2）。如果你知道 y 为 3，那么 x 有等同的机会为 1 或 2。类似地，如果 y 为 11，那么 x 要么为 5，要么为 6：

```
def random_x_given_y(y):
    if y <= 7:
        # 如果点数为7或以下的数，那么第一个骰子的点数有等同的机会为
        # 1, 2, …, (总点数 - 1)
        return random.randrange(1, y)
    else:
        # 如果点数为7或以上的数，那么第一个骰子的点数有等同的机会为
        # (total - 6), (总点数 - 5), …, 6
        return random.randrange(y - 6, 7)
```

吉布斯抽样的方法是先从任意（有效）的 x 和 y 值入手，然后不断用 y 条件下随机选择的 x 值替换原来的 x，并用 x 条件下随机选择的 y 值替换原来的 y。重复一定次数后，得到的 x 值和 y 值就可以作为根据无条件的联合分布获取的样本了：

```
def gibbs_sample(num_iters=100):
    x, y = 1, 2 # doesn't really matter
    for _ in range(num_iters):
        x = random_x_given_y(y)
        y = random_y_given_x(x)
    return x, y
```

通过下列代码你会发现，这种取样方法与直接取样的效果相似：

```
def compare_distributions(num_samples=1000):
    counts = defaultdict(lambda: [0, 0])
    for _ in range(num_samples):
        counts[gibbs_sample()][0] += 1
        counts[direct_sample()][1] += 1
    return counts
```

我们将在接下来的部分使用这种取样方法。

20.5　主题建模

在第 1 章我们建立“你应该知道的数据科学家”推荐系统的时候，我们只是根据科学家与读者的兴趣是否严格匹配来进行推荐的。

要想理解我们的用户，更高级的方法是识别这些兴趣背后的相关主题。有一种叫作隐含狄利克雷分析（latent dirichlet analysis，LDA）的技术，常用来确定一组文档的共同主题。我们会用它来分析包含每个用户的兴趣的那些文档。

这里的 LDA 与第 13 章中的朴素贝叶斯分类器有一些相似之处，它们都是用于处理文档的概率模型。我们将尽量避开一些数学细节问题，但对于该模型的某些假设却不得不说，如下所述。

- 存在固定数目的主题，即 K 个。
- 有一个给每个主题在单词上的概率分布赋值的随机变量。你可以把这个分布看作是单词 w 在给定主题 k 中出现的概率。
- 还有另一个随机变量来指出每个文档在主题下面的概率分布。你可以将这个分布看作是文档 d 中各主题所占比重。
- 文档中各个单词的生成方式为，首先（根据文档的主题分布情况）随机选择一个主题，然后（根据该主题下面各单词的分布情况）随机选择一个单词。

特别地，我们要建立一个文档（`documents`）集合，其中每个文档都是一个单词的列表。

同时，我们还要建立一个相应的 `document_topics` 集合，以便给每个文档中的每个单词都指定一个主题（这里用 0 到 K–1 之间的一个数字表示）。

这样的话，第 4 个文档中的第 5 个单词就可以表示为：

```
documents[3][4]
```

而这个选定的单词对应的主题可以表示为：

```
document_topics[3][4]
```

这非常显式地定义了每个文档在各个主题上的分布情况，同时也隐式地定义了每个主题在各个单词上面的分布情况。

我们可以估算出主题 1 产生一个特定单词的可能性，方法是将主题 1 产生该单词的次数除以主题 1 产生任意单词的次数。（类似地，我们在第 13 章中建立垃圾邮件过滤器时，也曾经用每个单词出现在垃圾邮件中的次数与这些单词出现在垃圾邮件中的总次数进行过相应的比较。）

这些主题都只是些数字，不过，我们可以用其权重最大的单词给它们取一个描述性的名称。接下来，我们只需设法生成 document_topics 即可。这时，吉布斯取样技术就派上用场了。

首先，我们以完全随机的方式给所有文档中的每个单词都赋予一个主题。现在，我们就以每次一个单词的方式遍历所有文档。对于给定的单词和文档，我们需要根据该文档中主题（当前）的分布情况以及相对于该主题各单词（当前）的分布情况来建立相应的权重。然后，我们会使用这些权重给这个单词选取新主题。如果将该过程迭代多次，我们就能利用主题 - 单词分布和文档 - 主题分布完成联合取样。

首先，我们需要一个函数，根据任意一组权重来随机选择一个索引：

```
def sample_from(weights):
    """returns i with probability weights[i] / sum(weights)"""
    total = sum(weights)
    rnd = total * random.random()       # 在0和总数之间均匀分布
    for i, w in enumerate(weights):
        rnd -= w                        # 返回最小的i
        if rnd <= 0: return i           # 因此weights[0] + … + weights[i] >
```

例如，如果你给它的权重为 [1, 3, 1]，那么五分之一的时间将返回 0，五分之一的时间将返回 1，同时还有五分之三的时间将返回 2。

我们的文档包含的是用户的各种兴趣，看起来可能像下面这样：

```
documents = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"]
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intellige
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

我们将设法找出 4 个主题，即 `K = 4`。

为了计算抽样权重，我们需要明确几项计数。下面，我们先给它们创建相应的数据结构。

我们要统计每个文档中每个主题出现的次数，代码如下：

```
# 计数的一个列表，每个文档各有一个列表
document_topic_counts = [Counter() for _ in documents]
```

我们要统计每个主题中每个单词出现的次数，代码如下：

```
# 计数的一个列表，每个主题各有一个列表
topic_word_counts = [Counter() for _ in range(K)]
```

我们要知道每个主题中单词的总数，代码如下：

```
# 数字的一个列表， 每个主题各有一个列表
topic_counts = [0 for _ in range(K)]
```

下面的代码统计每个文档中的单词总数：

```
# 数字的一个列表，每个文档各有一个列表
document_lengths = map(len, documents)
```

下面代码统计不同单词的数量：

```
distinct_words = set(word for document in documents for word in document)
W = len(distinct_words)
```

另外，下列代码可以用来统计文档的数量：

```
D = len(documents)
```

一旦掌握了这些数据，我们就可以了解（比如说）documents[3] 中与主题 1 相关的单词的数量，具体代码如下所示：

```
document_topic_counts[3][1]
```

同时，我们还可以找出与主题 2 相关的单词 nlp 出现的次数，具体代码如下所示：

```
topic_word_counts[2]["nlp"]
```

现在，我们已经为定义条件概率函数做好了准备。就像在第 13 章中那样，这里的每个主题和单词都需要有一个平滑项，来确保每个主题在任何文档中被选中的几率都不能为 0，同时保证每个单词在任何主题中被选中的几率也都不能为 0：

```
def p_topic_given_document(topic, d, alpha=0.1):
    """the fraction of words in document _d_
    that are assigned to _topic_ (plus some smoothing)"""

    return ((document_topic_counts[d][topic] + alpha) /
            (document_lengths[d] + K * alpha))

def p_word_given_topic(word, topic, beta=0.1):
    """the fraction of words assigned to _topic_
    that equal _word_ (plus some smoothing)"""

return ((topic_word_counts[topic][word] + beta) /
        (topic_counts[topic] + W * beta))
```

然后利用下列代码给更新中的主题确定权重：

```
def topic_weight(d, word, k):
    """given a document and a word in that document,
    return the weight for the kth topic"""
    return p_word_given_topic(word, k) * p_topic_given_document(k, d)

def choose_new_topic(d, word):
    return sample_from([topic_weight(d, word, k)
                        for k in range(K)])
```

上面的 topic_weight 之所以如此定义，背后是有坚实的数学理论作为依据的，不过其中的数学细节已经超出了本书的讨论范围。不过从直观的角度来看，如果已知一个单词和所在文档，那么该单词属于某主题的概率取决于两个方面，即该主题属于该文档的可能性以及该单词属于该主题的可能性。

这就是我们所需要的全部零部件。下面，我们开始将每个单词随机指派给一个话题，并计入相应的计数器：

```
random.seed(0)
document_topics = [[random.randrange(K) for word in document]
                   for document in documents]

for d in range(D):
    for word, topic in zip(documents[d], document_topics[d]):
        document_topic_counts[d][topic] += 1
        topic_word_counts[topic][word] += 1
        topic_counts[topic] += 1
```

我们的目标是利用主题 - 单词分布和文档 - 主题分布进行联合采样。为此，我们可以通过之前定义的条件概率进行吉布斯采样，具体代码如下所示：

```
for iter in range(1000):
    for d in range(D):
        for i, (word, topic) in enumerate(zip(documents[d],
                                              document_topics[d])):

            # 从计数中移除这个单词/主题
            # 以便它不会影响权重
            document_topic_counts[d][topic] -= 1
            topic_word_counts[topic][word] -= 1
```

```
        topic_counts[topic] -= 1
        document_lengths[d] -= 1

        # 基于权重选择一个新的主题
        new_topic = choose_new_topic(d, word)
        document_topics[d][i] = new_topic

        # 现在把它重新加到计数中
        document_topic_counts[d][new_topic] += 1
        topic_word_counts[new_topic][word] += 1
        topic_counts[new_topic] += 1
        document_lengths[d] += 1
```

这些主题都是什么呢？它们只是些数字而已：0、1、2 和 3。如果我们要让它们拥有名称的话，必须亲自给它们取名。下面，让我们来找出权重较大的 5 个单词（见表 20-1），具体代码如下所示：

```
for k, word_counts in enumerate(topic_word_counts):
    for word, count in word_counts.most_common():
        if count > 0: print k, word, count
```

表**20-1**：每个主题中最常见的单词

主题[illegible]			
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]	[illegible]

根据这些数据，我们就可以给主题取名了：

```
topic_names = ["Big Data and programming languages",
               "Python and statistics",
               "databases",
               "machine learning"]
```

至此我们就清楚模型是如何将主题分配到每个用户的兴趣上面了：

```
for document, topic_counts in zip(documents, document_topic_counts):
    print document
    for topic, count in topic_counts.most_common():
        if count > 0:
```

```
        print topic_names[topic], count,
print
```

其输出结果如下所示：

```
['Hadoop', 'Big Data', 'HBase', 'Java', 'Spark', 'Storm', 'Cassandra']
Big Data and programming languages 4 databases 3
['NoSQL', 'MongoDB', 'Cassandra', 'HBase', 'Postgres']
databases 5
['Python', 'scikit-learn', 'scipy', 'numpy', 'statsmodels', 'pandas']
Python and statistics 5 machine learning 1
```

如此等等。假使我们被要求用“ands”作为某个主题的名称，那么我们很可能需要使用更多的主题，尽管更可能的情况是我们没有足够的数据来学习。

20.6　延伸学习

- Natural Language Toolkit（http://www.nltk.org/）是 Python 语言一个流行（和非常全面）的 NLP 工具库。对于这个库，已经有专门的书籍（http://www.nltk.org/book/）对它进行全面的介绍，并且该书可以在线阅读。

- gensim（http://radimrehurek.com/gensim/）是一个用于主题建模的 Python 库，与从头开始建模相比，使用它来建模是一种更靠谱的途径。

第 21 章　网络分析

你跟周遭事物的所有联系定义了你是谁。

——亚伦 · 奥康奈尔

当我们面对许多有趣的数据问题时，如果把它们看作由各种类型的节点（node）和连接它们的边（edge）所构成的网络，那就再合适不过了。

例如，你的 Facebook 好友可以看作一个网络中的节点，而连接节点的边可以看作朋友关系。另外一个不太明显的例子是万维网本身，其中的每一个网页都是一个网络节点，而从一个页面到另一个页面的超链接则是这个网络的边。

Facebook 中的朋友关系都是相互的，也就是说，如果我是你的 Facebook 好友，那么你也肯定是我的好友。这样的话，我们就可以说这种网络中的边是无方向的（undirected）。但是，超链接却并非如此，即如果我的网站链接到白宫的网站，并不表示白宫的网站也会链接到我的网站。我们称这种类型的边为有方向的（directed）。

我们下面会介绍这两种类型的网络。

21.1　中介中心度

在第 1 章中，我们只能通过每位用户的好友数量来计算 DataSciencester 网络中的关键联系人。现在，我们已经有足够多的手段来寻找其他的计算方法了。下面，我们先来看看网络中的用户，具体如图 21-1 所示：

```
users = [
    { "id": 0, "name": "Hero" },
    { "id": 1, "name": "Dunn" },
    { "id": 2, "name": "Sue" },
    { "id": 3, "name": "Chi" },
    { "id": 4, "name": "Thor" },
    { "id": 5, "name": "Clive" },
    { "id": 6, "name": "Hicks" },
    { "id": 7, "name": "Devin" },
    { "id": 8, "name": "Kate" },
    { "id": 9, "name": "Klein" }
```

```
]
```

以及用户之间的好友关系：

```
friendships = [(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (3, 4), (4, 5), (5,
```

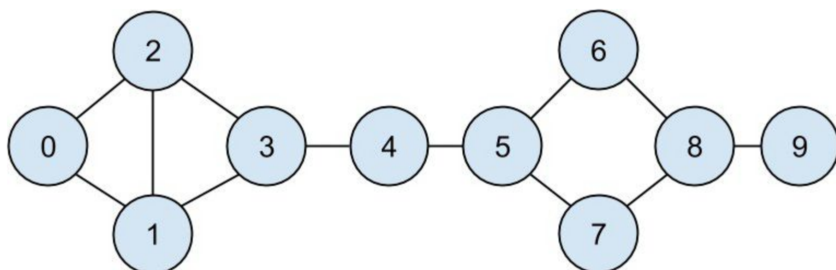


图 **21-1**：**DataSciencester** 网络

此外，我们还给每个用户的 dict 结构添加了相应的朋友列表：

```
for user in users:
    user["friends"] = []

for i, j in friendships:
    # 这能奏效是因为users[i]是id为i的用户
    users[i]["friends"].append(users[j]) # 添加i作为j的朋友
    users[j]["friends"].append(users[i]) # 添加j作为i的朋友
```

当时，我们对度中心性（degree centrality）的概念不甚满意，因为它与网络中直观展现在我们面前的关键联系人不甚相符。

另一种度量指标是中介中心度（betweenness centrality），它可以用来找出经常位于其他节点对之间的最短路径中的人。具体而言，中介中心度可以通过累加节点 j 和节点 k 之间经过节点 i 的最短路径所占比例，以及节点 j 和 k 之外所有的节点对中相应的比例来求出。

也就是说，如果我们想计算出 Thor 的中介中心度，首先要计算 Thor 之外的所有用户对之间的最短路径，然后再统计有多少条最短路径通过了 Thor 这个节点。比如说，Chi（`id` 为 3) 和 Clive（`id` 为 5) 之间唯一的最短路径经过了 Thor 这个节点，而 Hero（`id` 为 0) 和 Chi（`id` 为 3) 之间的两条最短路径则都没有经过 Thor。

因此，作为第一步，我们需要找出所有用户对之间的最短路径。不过，尽管存在许多可以高效计算最短路径的尖端算法，但是按照我们的惯例，这里将采用效率虽低一些但更加容易理解的算法。

这个算法（一个广度优先搜索的实现）在本书中算是比较复杂的一种，所以我们仔细探讨。

1. 我们的目标是建立一个以 `from_user` 为输入的函数，它能够找出到达其他每个用户的所有最短路径。

2. 我们将通过用户 ID 组成的列表来表示路径。由于每条路径的第一个节点总是 `from_user`，因此我们可以在这个列表中将该 ID 忽略。也就是说，这个代表路径的列表的长度等于该路径本身的长度。

3. 我们将维护一个名为 `shortest_paths_to` 的字典，其键为用户 ID，其值为以该用户 ID 结尾的路径构成的列表。如果最短路径是唯一的，那么这个列表就只包含一个路径。如果有多条最短路径的话，那么该列表将包含所有这些路径。

4. 我们还将维护一个名为 `frontier` 的队列来存放那些待考察的用户，并且它们的存放顺序就是相应的考察顺序。我们将以用户对的形式——即 `(prev_user, user)`——来进行存储，这样就能了解我们是如何到达每一个用户的。这个队列是通过 `from_user` 的所有相邻节点进行初始化的。（当然，我们之前从没有讨论过队列，其实它是专门为“在后端进行插入”操作和“在前端进行删除”操作经过优化的数据结构。在 Python 语言中，队列是由 `collections.deque` 模块来实现的，实际上它是一种双向队列。）

5. 当我们在图中进行探索的时候，每当发现新的邻居节点，只要还不知道通向它们的最短路径，就将它们添加到队尾以供将来进一步探索，并且以当前用户作为其 `prev_user`。

6. 当我们把一个用户从队列中删除时，如果之前从未遇到过该用户，那么我们肯定是找到了一个或多个通向它的最短路径：沿着到达 `prev_user` 的每个最短路径再走一步即是。

7. 当我们从队列中删除一个之前遇到过的用户时，我们不是找到了另一个最短路径（这种情况下我们应该将其添加到队尾），就是找到了一个更长的路径（这种情况下不用将其插入队尾）。

8. 当队列中已经没有用户时，说明我们已经搜遍了整个图（或者至少

也是起始用户所能够达到的部分），这时我们就可以停止了。

我们可以将这些步骤放入一个（大型）函数中，代码如下所示：

```
from collections import deque

def shortest_paths_from(from_user):

# 一个由"user_id"到该用户所有最短路径的字典
shortest_paths_to = { from_user["id"] : [[]] }

# 我们需要检查的(previous user, next user)队列
# 从所有(from_user, friend_of_from_user)对开始着手

frontier = deque((from_user, friend)
                 for friend in from_user["friends"])

# 直到队列为空为止
while frontier:

    prev_user, user = frontier.popleft()   # 删除该用户
    user_id = user["id"]                   # 即队列中的第一个用户

    # 若要向队列添加内容
    # 我们必须知道通向prev_user的某些最短路径
    paths_to_prev_user = shortest_paths_to[prev_user["id"]]
    new_paths_to_user = [path + [user_id] for path in paths_to_prev_user]

    # 我们可能已经知道了一条最短路径
    old_paths_to_user = shortest_paths_to.get(user_id, [])

    # 到目前为止，我们看到的到达这里的最短路径有多长？
    if old_paths_to_user:
        min_path_length = len(old_paths_to_user[0])
    else:
        min_path_length = float('inf')

    # 只留下那些刚找到的不太长的路径
    new_paths_to_user = [path
                         or path in new_paths_to_user
                         f len(path) <= min_path_length
                         nd path not in old_paths_to_user]

    shortest_paths_to[user_id] = old_paths_to_user + new_paths_to_user

    # 将这些从未谋面的"邻居"添加到frontier中
    frontier.extend((user, friend)
                    for friend in user["friends"]
                    if friend["id"] not in shortest_paths_to)

return shortest_paths_to
```

现在我们可以将这些 dict 存放到各个节点中了：

```
for user in users:
    user["shortest_paths"] = shortest_paths_from(user)
```

好了，现在终于可以计算中介中心度了。对于任意一对节点 i 和 j，我们都知道从 i 到 j 有 n 条最短路径。然后，对应于每一条这样的最短路径，我们只要给该路径中的每个节点的中心度加 $1/n$ 即可：

```
for user in users:
    user["betweenness_centrality"] = 0.0

for source in users:
    source_id = source["id"]
    for target_id, paths in source["shortest_paths"].iteritems():

        if source_id < target_id:       # 不要加倍计数
            num_paths = len(paths)      # 有多少最短路径
            contrib = 1 / num_paths     # 中心度加1/n
            for path in paths:
                for id in path:
                    if id not in [source_id, target_id]:
                        users[id]["betweenness_centrality"] += contrib
```

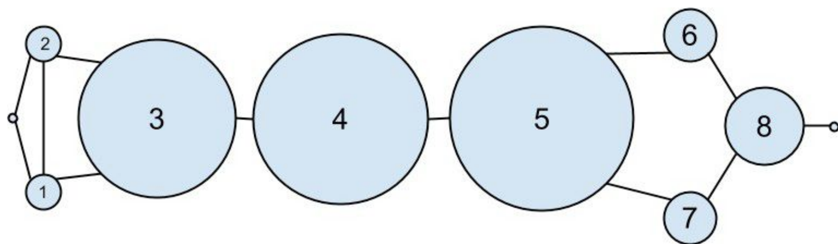


图 **21-2**：根据中介中心度的大小绘制的 **DataSciencester** 网络

如图 21-2 所示，用户 0 和 9 的中心度为 0（因为它们不在其他用户之间的任何一条最短路径上），而用户 3、4 和 5 则都具有较高的中心度（因为它们都位于多条最短路径上）。



一般来说，中心度的数值本身并不具有多大意义。我们关心的是每个节点的中心度数值与其他节点的相对大小。

此外，还有一个需要关注的衡量指标，即所谓的接近中心度

（closeness centrality）。首先，为每个用户计算其疏远度（farness），即该用户到所有其他用户的最短路径的长度总和。

由于我们已经计算出每一对节点之间的最短路径，因此，只要对其求和即可。（如果有多个最短路径，并且都具有相同的长度，那么我们只考察第一个即可。）

```
def farness(user):
    """the sum of the lengths of the shortest paths to each other user"""
    return sum(len(paths[0])
               for paths in user["shortest_paths"].values())
```

这样一来，接近中心度的计算量就很小了（见图 21-3）：

```
for user in users:
    user["closeness_centrality"] = 1 / farness(user)
```

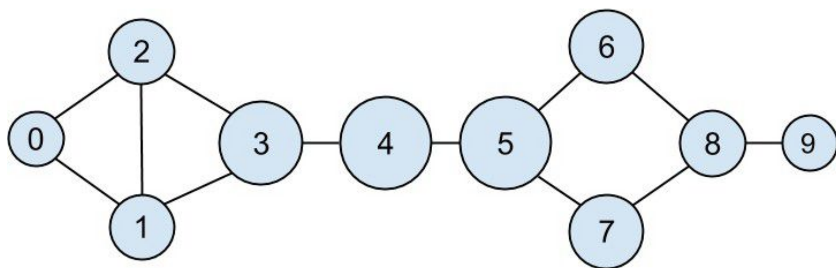


图 **21-3**：根据接近中心度绘制的 **DataSciencester** 网络

我们看到，尽管非常靠近中心的节点离外围节点很远，但是图中各节点在大小上面差别不是很大。

正如我们所看到的那样，计算最短路径是一件苦差事。因此，中介中心度和接近中心度很少用于大型网络。还有一种不太直观的特征向量中心度（eigenvector centrality），由于计算起来更容易，所以更加常用。

21.2　特征向量中心度

要想介绍特征向量中心度，我们就不得不讨论特征向量，而要想讨论特征向量，我们就必须介绍矩阵乘法。

21.2.1 矩阵乘法

假设 A 是一个 $n_1 \times k_1$ 矩阵，B 是一个 $n_2 \times k_2$ 矩阵，并且 $k_1 = n_2$，那么这两个矩阵的乘积 AB 则是一个 $n_1 \times k_2$ 矩阵，其中第 (i,j) 项为：

$$A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{ik}B_{kj}$$

下面，我们仅计算矩阵 A 的第 i 行（可以视为一个向量）与矩阵 B 的第 j 列（也可以视为 一个向量）的点积，具体代码如下所示：

```
def matrix_product_entry(A, B, i, j):
    return dot(get_row(A, i), get_column(B, j))
```

之后，我们就可以通过下列代码实现矩阵的乘法运算了：

```
def matrix_multiply(A, B):
    n1, k1 = shape(A)
    n2, k2 = shape(B)
    if k1 != n2:
        raise ArithmeticError("incompatible shapes!")

    return make_matrix(n1, k2, partial(matrix_product_entry, A, B))
```

请注意，如果 A 是一个 $n \times k$ 矩阵，B 是一个 $k \times 1$ 矩阵，那么 AB 则为 $n \times 1$ 矩阵。如果我们将一个向量看作是一维矩阵，就可以把 A 看作是将 k 维向量映射为 n 维向量的一个函数，实际上这个函数就是矩阵乘法。

之前，我们将向量简单地表示为列表，实际上两者之间是不能完全划等号的：

```
v = [1, 2, 3]
v_as_matrix = [[1],
               [2],
               [3]]
```

因此，我们需要定义相应的辅助函数，以便实现两种表示形式之间的转换：

```
def vector_as_matrix(v):
```

```
    """returns the vector v (represented as a list) as a n x 1 matrix"""
    return [[v_i] for v_i in v]

def vector_from_matrix(v_as_matrix):
    """returns the n x 1 matrix as a list of values"""
    return [row[0] for row in v_as_matrix]
```

如此一来，我们就可以利用 matrix_multiply 来定义矩阵运算了：

```
def matrix_operate(A, v):
    v_as_matrix = vector_as_matrix(v)
    product = matrix_multiply(A, v_as_matrix)
    return vector_from_matrix(product)
```

当 A 是一个方阵时，此操作会将 n 维向量映射为另一个 n 维向量。对于某矩阵 A 和向量 v，对向量 v 进行 A 变换有时候会等效于用一个标量来乘向量 v，即所得到的向量与 v 同向。当发生这种情况（且 v 不是零向量）时，我们称 v 为 A 的特征向量。同时，我们称这个乘数为特征值（eigenvalue）。

确定矩阵 A 的特征向量的一种可行方法是取一个随机向量 v，然后利用 `matrix_operate` 对其进行调整，从而得到一个长度为 1 的向量，重复该过程直到收敛为止：

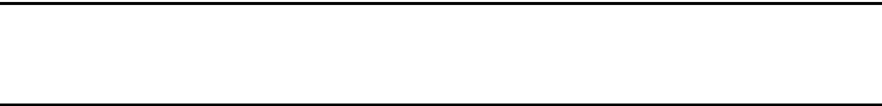
```
def find_eigenvector(A, tolerance=0.00001):
    guess = [random.random() for __ in A]

while True:
    result = matrix_operate(A, guess)
    length = magnitude(result)
    next_guess = scalar_multiply(1/length, result)

    if distance(guess, next_guess) < tolerance:
        return next_guess, length   # eigenvector, eigenvalue
        guess = next_guess
```

通过这种构造方法返回的向量 `guess` 将具备这样的特点：当你对它应用 `matrix_operate` 函数并将其长度缩为 1 的时候，得到的向量与其自身极为接近。这就意味着它是一个特征向量。

请注意，并不是所有的实数矩阵都具有特征向量和特征值。例如，请看下列矩阵：



```
rotate = [[ 0, 1],
          [-1, 0]]
```

上述代码的作用是按照顺时针方向将向量旋转 90 度，这意味着，对于这个矩阵来说，只有一个向量能够映射到自身的数乘上面，这个向量就是零向量。如果你执行 find_eigenvector(rotate)，它会永远运行下去。即使是具备特征向量的矩阵，有时候也会陷入这种死循环。请看下面的矩阵：

```
flip = [[0, 1],
        [1, 0]]
```

对于任意向量 [x, y]，这个矩阵都会将其映射为 [y, x]。这就意味着，[1, 1] 是一个特征值为 1 的特征向量。但是，如果你从一个 x 和 y 并不相等的随机向量着手的话，那么，find_eigenvector 将会来回交换这两个坐标值，并且永远也不会停下来。（Not-from-scratch 是一个类似于 NumPy 的 Python 库，由于它采用了不同的处理方法，因此能够有效处理这种情形。）尽管如此，只要 find_eigenvector 能返回一个结果，那么这个结果肯定是一个特征向量。

21.2.2　中心度

该如何利用特征向量来帮助我们理解 DataSciencester 网络呢？

首先，我们需要用 adjacency_matrix 来表示网络中的连接，其中第 (i,j) 个元素的值要么为 1（如果用户 i 和用户 j 是朋友的话），要么为 0（如果他们不是朋友的话）：

```
def entry_fn(i, j):
    return 1 if (i, j) in friendships or (j, i) in friendships else 0

n = len(users)
adjacency_matrix = make_matrix(n, n, entry_fn)
```

对于每个用户来说，他的特征向量中心度就是在 find_eigenvector 返回的本征向量中的该用户对应的那个元素（见图 21-4）：



具体技术细节这里就不多说了，大家只要知道任

何非零的邻接矩阵必然有一个所有的值皆非负的特征向量就行了。幸运的是，我们只要借助于 find_eigenvector 函数就能够找到这种 adjacency_matrix。

```
eigenvector_centralities, _ = find_eigenvector(adjacency_matrix)
```

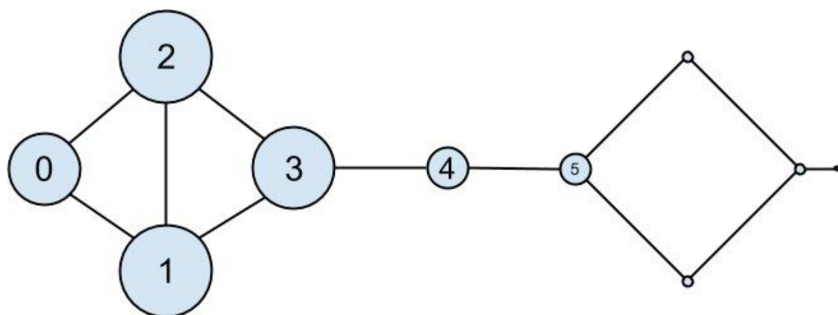


图 21-4：根据特征向量中心度绘制的 **DataSciencester** 网络

特征向量中心度较高的用户，不仅会拥有较多的连接，而且还倾向于连接到具有较高中心度的那些人。

就上图而言，用户 1 和用户 2 具有最高的中心度，这是由于他们两个都有三条连接是通向具有高中心度的对方的。如果我们将其移除，他们的中心度就会直线下降。

在这种小型的网络上中，特征向量中心度的行为会有些怪异。当你尝试增减连接的时候，你会发现，只要对网络进行稍微的修改，中心度的数值就会发生戏剧性的变化。对于比较大型的网络来说，这种情况就不太明显。

我们仍然没有介绍为什么特征向量能够较好地度量中心度。这是因为特征向量意味着，如果你计算：

```
matrix_operate(adjacency_matrix, eigenvector_centralities)
```

其结果就等于用一个标量去乘以 eigenvector_centralities。

如果你了解矩阵乘法的运算机制，就会知道 matrix_operate 求出的向量的第 i 个元素为：

```
dot(get_row(adjacency_matrix, i), eigenvector_centralities)
```

这实际上就是对连接到用户 i 的各个用户的特征向量中心度进行求和。

换句话说，特征向量中心度就是一些数值，即每个用户对应一个数值，而每个用户的值就是他的相邻值之和的固定倍数。在这种情况下，中心度就意味着要跟处于中心地位的人交朋友。你结交的人的中心度越高，你的中心度也就越高。当然，这明显是一个循环定义，而特征向量就是打破这个循环的突破口。

对此还有另外一种理解方法，那就是考察 `find_eigenvector` 函数的处理方式。它首先给每个节点随机指定一个中心度，然后重复以下两个步骤，直到这个过程收敛为止。

1. 赋予每个节点一个新的中心度分数，该分数等于该节点相邻节点的（原）中心度分数之和。

2. 调整中心度向量，直到其大小变成 1 为止。尽管这种做法包含的数学原理有点让人摸不着头脑，但是就计算本身而言，还是相当简单的（这一点与中介中心度不同），同时，它也非常适用于巨型网络图。

21.3　有向图与PageRank

由于 DataSciencester 没有获得人们的热烈追捧，因此，负责营收的副总决定将网站从交友模式转换为赞助模式。事实证明，除了高科技业的猎头非常关心哪些数据科学家备受其他数据科学家推崇之外，没人对科学家之间的好友关系特别在意。

在这个新的模型中，我们所关注的赞助 `(source, target)` 并不表示互反关系，而是表示用户 `source` 认为用户 `target` 是一位令人惊畏的数据科学家（见图 21-5）。因此，我们需要考虑这种不对称性：

```
endorsements = [(0, 1), (1, 0), (0, 2), (2, 0), (1, 2),
                (2, 1), (1, 3), (2, 3), (3, 4), (5, 4),
                (5, 6), (7, 5), (6, 8), (8, 7), (8, 9)]

for user in users:
    user["endorses"] = []          # 增加一个列表来追踪外方的赞助
    user["endorsed_by"] = []       # 增加另外一个列表来追踪赞助
```

```
for source_id, target_id in endorsements:
    users[source_id]["endorses"].append(users[target_id])
    users[target_id]["endorsed_by"].append(users[source_id])
```

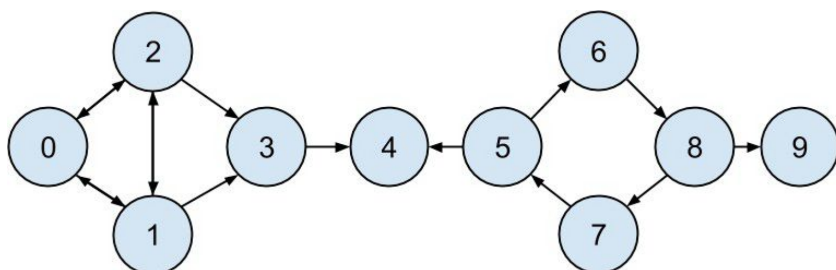


图 **21-5**：基于赞助关系的 **DataSciencester** 网络

这样的话，我们就能够轻而易举地找出 `most_endorsed`（最受推崇的）数据科学家，从而将这些信息出售给猎头们：

```
endorsements_by_id = [(user["id"], len(user["endorsed_by"]))
                      for user in users]

sorted(endorsements_by_id,
       key=lambda (user_id, num_endorsements): num_endorsements,
       reverse=True)
```

然而，"赞同票数"这种指标是很容易被人搞鬼的。实际上，你只要创建大量傀儡账户，然后让这些账户给你投票就行了。或者，你还可以跟朋友们商量好，都彼此捧场也行。（例如用户 0、1 和 2 好像就是这么干的。）

因此，指标最好还要考虑到给你投赞同票的那些人。也就是说，来自得票数较多的人的投票的分量应该重于得票数较少的那些人的投票。这实际上就是 PageRank 算法的思想精华，Google 就是利用它来给网站排名的，主要考量的就是链接到该网站的其他站点、到达该网站的链接等。

（这是否让你想起了特征向量中心度背后的思想依据呢？）

下面是这种思想的简化版本。

1. 网络中 PageRank 的总分数为 1（或 100%）。

2. 最初，这个 PageRank 被均匀分布到网络的各个节点中。

3. 在每一步中，每个节点的 PageRank 很大一部分将均匀分布到其外部链接中。

4. 在每个步骤中，每个节点的 PageRank 的其余部分被均匀地分布到所有节点上。

```
def page_rank(users, damping = 0.85, num_iters = 100):

    # 一开始均匀分布PageRank
    num_users = len(users)
    pr = { user["id"] : 1 / num_users for user in users }

    # 这是PageRank的一小部分
    # 每个节点进行各自的迭代
    base_pr = (1 - damping) / num_users

    for __ in range(num_iters):
        next_pr = { user["id"] : base_pr for user in users }
        for user in users:
            # 将PageRank分布到外部链接中
            links_pr = pr[user["id"]] * damping
            for endorsee in user["endorses"]:
                next_pr[endorsee["id"]] += links_pr / len(user["endorses"]
        pr = next_pr

    return pr
```

PageRank（见图 21-6）表明，用户 4（也就是 Thor）是排名最高的数据科学家。

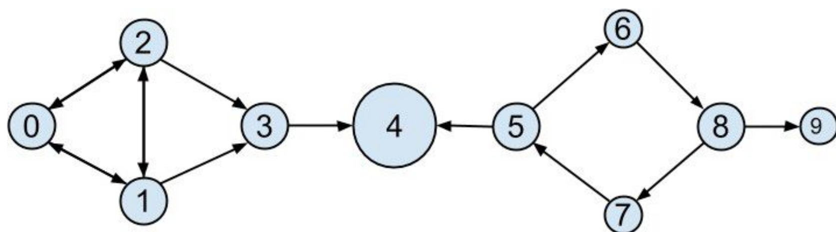


图 **21-6**：利用 **PageRank** 绘制的 **DataSciencester** 网络

与用户 0、1 和 2 相比，虽然给他投票的人（2 个）并不多，但是他的得票数还要考虑投票方自身的排名。此外，两个投票方都给只给他投了票，这就意味着他不必与别人分享他们的排名。

21.4 延伸学习

- 除了本文介绍的这些中心度指标外，还有许多其他不同的指标（https://en.wikipedia.org/wiki/Centrality），不过这里所介绍的都是些最常见的指标。

- NetworkX（http://networkx.github.io/）是一个用于网络分析的 Python 库。它为我们提供了许多函数，来帮助计算中心度以及实现图的可视化。

- Gephi（http://gephi.github.io/）是一个让人爱恨交织的基于图形用户界面的网络可视化工具。

第 22 章　推荐系统

哦，老天，老天，你为什么如此随便，老是把提供错误建议的人送到这个世间？！

——亨利 · 菲尔丁

现实中，利用数据提供某种建议也是很常见的。例如，Netflix 能够向用户推荐他们可能想看的电影，亚马逊会向你推荐你可能会买的商品，Twitter 会为你推荐你可能想关注的用户。本章将介绍几种利用数据来提供建议的方法。

需要指出的是，这里要考察的 `users_interests` 是之前就曾用过的一个数据集：

```
users_interests = [
    ["Hadoop", "Big Data", "HBase", "Java", "Spark", "Storm", "Cassandra"]
    ["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"],
    ["Python", "scikit-learn", "scipy", "numpy", "statsmodels", "pandas"],
    ["R", "Python", "statistics", "regression", "probability"],
    ["machine learning", "regression", "decision trees", "libsvm"],
    ["Python", "R", "Java", "C++", "Haskell", "programming languages"],
    ["statistics", "probability", "mathematics", "theory"],
    ["machine learning", "scikit-learn", "Mahout", "neural networks"],
    ["neural networks", "deep learning", "Big Data", "artificial intellige
    ["Hadoop", "Java", "MapReduce", "Big Data"],
    ["statistics", "R", "statsmodels"],
    ["C++", "deep learning", "artificial intelligence", "probability"],
    ["pandas", "R", "Python"],
    ["databases", "HBase", "Postgres", "MySQL", "MongoDB"],
    ["libsvm", "regression", "support vector machines"]
]
```

同时，我们要考虑如何根据用户当前特定的兴趣来向其推荐新的感兴趣的东西。

22.1　手工甄筛

在互联网出现之前，如果你需要得到某些读书建议的话，恐怕得到图书馆去，那里的图书管理员通常能够根据你的兴趣或者你喜欢的书籍来推荐书籍。

鉴于 DataSciencester 的用户及其兴趣的数量有限，你只需花费一个下午的时间就能轻松通过人工方式向每个用户推荐他们感兴趣的东西。但这种方法并不是特别好，因为它受制于你的个人经验和想象力。（当然，这并不意味着我认为你的个人经验和想象力是有限的。）因此，我们要设法让数据来做这件事情。

22.2　推荐流行事物

一个比较简单的方法就是直接推荐比较流行的东西：

```
popular_interests = Counter(interest
                            for user_interests in users_interests
                            for interest in user_interests).most_common()
```

得到的结果可能像下面这样：

```
[('Python', 4),
 ('R', 4),
 ('Java', 3),
 ('regression', 3),
 ('statistics', 3),
 ('probability', 3),
 # …
]
```

完成上述计算后，我们就可以向用户推荐那些当前最流行的、他尚未感兴趣的东西：

```
def most_popular_new_interests(user_interests, max_results=5):
    suggestions = [(interest, frequency)
                   for interest, frequency in popular_interests
                   if interest not in user_interests]
    return suggestions[:max_results]
```

因此，如果你是用户 1，并且当前的兴趣为：

```
["NoSQL", "MongoDB", "Cassandra", "HBase", "Postgres"]
```

那么，我们会向你推荐下列内容：

```
```

```
most_popular_new_interests(users_interests[1], 5)

# [('Python', 4), ('R', 4), ('Java', 3), ('regression', 3), ('statistics',
```

如果你是用户 3，而且上面这些东西中有很多早就是你之前的兴趣所在了，那么你会得到下面的建议：

```
[('Java', 3),
 ('HBase', 3),
 ('Big Data', 3),
 ('neural networks', 2),
 ('Hadoop', 2)]
```

当然，"很多人对 Python 感兴趣，所以你也可能感兴趣"并非最具有说服力的推销口号。如果某人是我们网站的新人，我们对他一无所知，那么这可能就是最好的推荐方法了。下面让我们看看如何根据用户的兴趣来提供更好的建议。

22.3　基于用户的协同过滤方法

一种利用用户兴趣的方法是根据这些兴趣找到有类似爱好的人，然后再根据这些人的爱好来向你推荐你可能感兴趣的东西。

为此，我们需要找到一种指标来衡量两个用户之间的相似程度。就这里来说，我们所使用的指标叫作余弦相似度（cosine similarity）。给定两个向量 `v` 和 `w`，余弦相似度的定义如下所示：

```
def cosine_similarity(v, w):
    return dot(v, w) / math.sqrt(dot(v, v) * dot(w, w))
```

它用来测量 `v` 和 `w` 之间的"角度"。如果 `v` 和 `w` 指向同一个方向，那么分子和分母都相等，所以其余弦相似度等于 1。如果 `v` 和 `w` 指向相反的方向，那么其余弦相似度就等于 -1。如果 `v` 等于 0，那么无论 `w` 是否为 0（反之亦然），dot(`v`, `w`) 总是等于 0，所以余弦相似度总为 0。

我们会把这个计算方法应用到由 0 和 1 构成的向量上面，其中每个向量都表示一个用户的某种爱好。如果用户对第 i 项事物感兴趣的话，则 `v[i]` 取 1，否则为 0。因此，"爱好相似的用户"就意味着"兴趣

向量的方向几乎相同的用户”。兴趣完全相同的用户，其相似度为 1。而没有共同兴趣的用户，其相似度为 0。否则的话，其相似度会介于两者之间：该数值越接近 1，表示“越相似”，该数值越接近 0，表示“越不相似”。

通常来说，从收集已知的兴趣并为其（隐式）指定索引着手是一个不错的主意。为此，我们可以用集合的观点将那些不重复的兴趣收集到一起，然后把它们放到一个列表中，并对其进行排序。这样的话，在这个列表中的第一个兴趣就是兴趣 0，其他以此类推：

```
unique_interests = sorted(list({ interest
                                 for user_interests in users_interests
                                 for interest in user_interests }))
```

我们将得到一个列表，开头部分如下所示：

```
['Big Data',
 'C++',
 'Cassandra',
 'HBase',
 'Hadoop',
 'Haskell',
 # …
]
```

接下来，我们要给每个用户生成一个由 0 和 1 组成的“兴趣”向量。为此，我们只需遍历 `unique_interests` 列表，如果用户有某种兴趣，则相应元素置 1，否则置 0：

```
def make_user_interest_vector(user_interests):
    """given a list of interests, produce a vector whose ith element is 1
    if unique_interests[i] is in the list, 0 otherwise"""
    return [1 if interest in user_interests else 0
            for interest in unique_interests]
```

之后，我们还可以创建用户兴趣矩阵，为此，我们只需将这个函数映射到由用户的兴趣构成的列表的列表上面即可：

```
user_interest_matrix = map(make_user_interest_vector, users_interests)
```

现在，如果用户 `i` 对兴趣 `j` 感兴趣，则

user_interest_matrix[i][j] 等于 1，否则为 0。

由于我们的数据集非常小，因此所有用户两两之间的相似性的计算量不是很大：

```
user_similarities = [[cosine_similarity(interest_vector_i, interest_vector_j)
                      for interest_vector_j in user_interest_matrix]
                     for interest_vector_i in user_interest_matrix]
```

在这之后，user_similarities[i][j] 的数值就能够告诉我们用户 i 和用户 j 之间的相似度。

举例来说，user_similarities[0][9] 等于 0.57，因为这两个用户都对 Hadoop、Java 和 Big Data 感兴趣。同时，user_similarities[0][8] 的值只有 0.19，因为用户 0 和用户 8 只有一个共同的兴趣，即 Big Data。

就 user_similarities[i] 而言，它存放的是用户 i 相对于所有用户的相似度。我们可以用它来写一个函数，来找出与给定用户最相似的用户。同时，我们还需要确保这里不包括用户自身以及相似度为 0 的那些用户。下面我们按照相似度从大到小的顺序对结果进行排序：

```
def most_similar_users_to(user_id):
    pairs = [(other_user_id, similarity)                      # 查找
             for other_user_id, similarity in                 # 其他用户
                enumerate(user_similarities[user_id])         # 非零
             if user_id != other_user_id and similarity > 0]  # 相似度

    return sorted(pairs,                                      # 将其排序
                  key=lambda (_, similarity): similarity,     # 相似度
                  reverse=True)                               # 由大到小
```

举例来说，如果我们调用函数 most_similar_users_to(0)，将得到下列输出：

```
[(9, 0.5669467095138409),
 (1, 0.3380617018914066),
 (8, 0.1889822365046136),
 (13, 0.1690308509457033),
 (5, 0.1543033499620919)]
```

那么，我们该如何利用这个结果向用户推荐新的兴趣呢？对于每种兴趣，我们可以将其他对其感兴趣的用户的用户相似度加起来：

```
def user_based_suggestions(user_id, include_current_interests=False):
    # 将相似度加起来
    suggestions = defaultdict(float)
    for other_user_id, similarity in most_similar_users_to(user_id):
        for interest in users_interests[other_user_id]:
            suggestions[interest] += similarity

    # 将它们转化成已排序的列表
    suggestions = sorted(suggestions.items(),
                         key=lambda (_, weight): weight,
                         reverse=True)

    # 并且（有可能）排除已存在的兴趣
    if include_current_interests:
        return suggestions
    else:
        return [(suggestion, weight)
                for suggestion, weight in suggestions
                if suggestion not in users_interests[user_id]]
```

如果我们调用 `user_based_suggestions(0)`，那么在推荐的兴趣中比较靠前的几个为：

```
[('MapReduce', 0.5669467095138409),
 ('MongoDB', 0.50709255283711),
 ('Postgres', 0.50709255283711),
 ('NoSQL', 0.3380617018914066),
 ('neural networks', 0.1889822365046136),
 ('deep learning', 0.1889822365046136),
 ('artificial intelligence', 0.1889822365046136),
 #...
]
```

这些东西对于自称对“Big Data”和数据库相关的主题感兴趣的人来说，看起来的确是相当不错的建议。（这些权重自身并无意义，它们只是用于进行排序。）

但是，当兴趣的数量很大的时候，这种方法就玩不转了。还记得第 12 章中介绍的维度灾难吗？在高维向量空间中，绝大多数向量之间都是离得非常远的（因此它们之间的方向也悬殊很大）。也就是说，当兴趣的数量变大时，即使是与给定用户“最相似的用户”，实际上也很可能根本没有相似之处。

想象一个类似亚马逊这样的网站，在过去几十年中，我已经从它那里购买了数以千计的商品。你可能试图通过购买模式来找出跟我类似的用户，但在这个世界上，除了我自己之外，恐怕没有谁的购买历史看起来更像我了。无论与我“最相似的”顾客是谁，他很可能根本就不像我，因此，如果将他购买的物品推荐给我的话，基本上就注定了这是一个糟糕的建议。

22.4　基于物品的协同过滤算法

还有一种方法，即直接计算两种兴趣之间的相似度，然后将与用户当前兴趣相似的兴趣放到一起，并从中为用户推荐感兴趣的东西。

首先，我们要对用户兴趣矩阵进行转置（transpose），以使行对应于兴趣，列对应于用户：

```
interest_user_matrix = [[user_interest_vector[j]
                         for user_interest_vector in user_interest_matrix]
                        for j, _ in enumerate(unique_interests)]
```

这么做之后，结果如何呢？这时，矩阵 `interest_user_matrix` 的行 `j` 就是矩阵 `user_interest_matrix` 的列 `j`。也就是说，1 表示用户有某种兴趣，0 表示用户没有某种兴趣。

举例来说，假设 `unique_interests[0]` 为 Big Data，那么 `interest_user_matrix[0]` 则为：

```
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0]
```

这是因为用户 0、8 和 9 都对 Big Data 感兴趣。

现在，我们可以再次利用余弦相似度。如果喜欢两个主题的用户完全重合，那么它们的相似度为 1。如果喜欢两个主题的用户没有一个是重合的，那么它们的相似度将是 0：

```
interest_similarities = [[cosine_similarity(user_vector_i, user_vector_j)
                          for user_vector_j in interest_user_matrix]
                         for user_vector_i in interest_user_matrix]
```

例如，我们可以通过下列代码找出与 Big Data（兴趣 0）最相似的

项：

```
def most_similar_interests_to(interest_id):
    similarities = interest_similarities[interest_id]
    pairs = [(unique_interests[other_interest_id], similarity)
             for other_interest_id, similarity in enumerate(similarities)
             if interest_id != other_interest_id and similarity > 0]
    return sorted(pairs,
                  key=lambda (_, similarity): similarity,
                  reverse=True)
```

下面是推荐的相似兴趣：

```
[('Hadoop', 0.8164965809277261),
 ('Java', 0.6666666666666666),
 ('MapReduce', 0.5773502691896258),
 ('Spark', 0.5773502691896258),
 ('Storm', 0.5773502691896258),
 ('Cassandra', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('neural networks', 0.4082482904638631),
 ('HBase', 0.3333333333333333)]
```

现在，我们可以通过总结与其兴趣相似的东西来为其提供建议：

```
def item_based_suggestions(user_id, include_current_interests=False):
    # 将相似的兴趣相加
    suggestions = defaultdict(float)
    user_interest_vector = user_interest_matrix[user_id]
    for interest_id, is_interested in enumerate(user_interest_vector):
        if is_interested == 1:
            similar_interests = most_similar_interests_to(interest_id)
            for interest, similarity in similar_interests:
                suggestions[interest] += similarity

# 根据权重将其排序
suggestions = sorted(suggestions.items(),
                     key=lambda (_, similarity): similarity,
                     reverse=True)

if include_current_interests:
    return suggestions
else:
    return [(suggestion, weight)
            for suggestion, weight in suggestions
            if suggestion not in users_interests[user_id]]
```

下面是为用户 0 提供的（看上去比较恰当的）建议：

```
[('MapReduce', 1.861807319565799),
 ('Postgres', 1.3164965809277263),
 ('MongoDB', 1.3164965809277263),
 ('NoSQL', 1.2844570503761732),
 ('programming languages', 0.5773502691896258),
 ('MySQL', 0.5773502691896258),
 ('Haskell', 0.5773502691896258),
 ('databases', 0.5773502691896258),
 ('neural networks', 0.4082482904638631),
 ('deep learning', 0.4082482904638631),
 ('C++', 0.4082482904638631),
 ('artificial intelligence', 0.4082482904638631),
 ('Python', 0.2886751345948129),
 ('R', 0.2886751345948129)]
```

22.5　延伸学习

- Crab（http://muricoca.github.io/crab/）是一个打造推荐系统的 Python 框架。
- Graphlab 也提供了一个推荐工具包（https://dato.com/products/create/docs/graphlab.toolkits.recommender.html）。
- Netflix Prize（http://www.netflixprize.com/）是一项比较著名的竞赛活动，旨在为 Netflix 用户打造更好的电影推荐系统。

第 23 章　数据库与 SQL

回忆，是最贴心的朋友，也是最可怕的敌人。

——吉尔伯特 · 帕克

我们使用的数据通常存储于数据库中。数据库是用来有效存储与查询数据的系统。绝大部分数据库属于关系型（relational）数据库，例如 Oracle、MySQL 与 SQL Server，它们把数据存储在表中，并专门通过结构化查询语言（SQL）来查询。SQL 是一种用来处理数据的声明性语言。

SQL 是数据科学家工具箱中相当重要的一部分。本章中我们来创建 NotQuiteABase——一个类似于数据库的 Python 实现。我们会学习 SQL 的基础知识，并且在我们的类数据库中展示它是如何工作的，这是让你从零基础开始理解它们工作原理的最佳方式。希望你通过解决 NotQuiteABase 中的问题，可以很好地理解在 SQL 中如何解决相同的问题。

23.1　CREATE TABLE与INSERT

关系型数据库是表（以及表之间的关系）的集合。表是行的简单集合，这与我们前面讨论过的矩阵不同。但是表有一个固定的结构，包含列名与列的类型。

例如，假设有一个数据集 `users`，对于每个用户它都包含三个属性 `user\_id`、`name` 和 `num\_friends`：

```
users = [[0, "Hero", 0],
         [1, "Dunn", 2],
         [2, "Sue", 3],
         [3, "Chi", 3]]
```

在 SQL 中，我们可以这样创建表：

```
CREATE TABLE users (
    user_id INT NOT NULL,
    name VARCHAR(200),
    num_friends INT);
```

注意，我们设定 user_id 和 num_friends 都必须是整数（且 user_id 不可以为 NULL，因为 NULL 表示缺失值，类似于我们的 None），并且名字必须是长度不大于 200 的字符串。NotQuiteABase 不考虑类型，但这里我们当作它会考虑。



SQL 几乎不区分大小写和缩进形式。本书中的大小写与缩进风格是我的个人习惯。如果你开始学习 SQL，很可能会遇上不同风格的例子。

你可以使用 INSERT 语句来插入行：

```
INSERT INTO users (user_id, name, num_friends) VALUES (0, 'Hero', 0);
```

注意，SQL 语句需要用分号结尾，并且字符串需要用单引号括住。

在 NotQuiteABase 中，只要简单地设定列名就能创建一个表。要想插入一个行，你可以使用表的 insert() 方法，这个方法使用一个包含行值的列表，行值需按照表的列名顺序排列。

从本质上说，我们存储的每行都类似于一个从列名到值映射的字典。一个真正的数据库不会使用这么浪费空间的方式，但这样做可以使 NotQuiteABase 更易于处理：

```
class Table:
    def __init__(self, columns):
        self.columns = columns
        self.rows = []

    def __repr__(self):
        """pretty representation of the table: columns then rows"""
        return str(self.columns) + "\n" + "\n".join(map(str, self.rows))

    def insert(self, row_values):
        if len(row_values) != len(self.columns):
            raise TypeError("wrong number of elements")
        row_dict = dict(zip(self.columns, row_values))
        self.rows.append(row_dict)
```

例如，我们可以创建：

```
users = Table(["user_id", "name", "num_friends"])
```

```
users.insert([0, "Hero", 0])
users.insert([1, "Dunn", 2])
users.insert([2, "Sue", 3])
users.insert([3, "Chi", 3])
users.insert([4, "Thor", 3])
users.insert([5, "Clive", 2])
users.insert([6, "Hicks", 3])
users.insert([7, "Devin", 2])
users.insert([8, "Kate", 2])
users.insert([9, "Klein", 3])
users.insert([10, "Jen", 1])
```

如果你打印 `users`，可以看到：

```
['user_id', 'name', 'num_friends']
{'user_id': 0, 'name': 'Hero', 'num_friends': 0}
{'user_id': 1, 'name': 'Dunn', 'num_friends': 2}
{'user_id': 2, 'name': 'Sue', 'num_friends': 3}
...
```

23.2 UPDATE

有时候你需要更新已经存在于数据库中的数据。例如，如果 Dunn 结识了一位新朋友，你就得这样做：

```
UPDATE users
SET num_friends = 3
WHERE user_id = 1;
```

其核心特征是：

- 哪张表需要更新
- 哪些行需要更新
- 哪些字段需要更新
- 新值应该是什么

我们将为 NotQuiteABase 增加类似的 `update` 方法。第一个语句是 `dict`，其键是需要更新的列，其值是这些字段的新值。第二个语句是 `predicate`，它对需要更新的行返回 `True`，否则返回

False：

```
def update(self, updates, predicate):
    for row in self.rows:
        if predicate(row):
            for column, new_value in updates.iteritems():
                row[column] = new_value
```

然后我们只需这样做：

```
users.update({'num_friends' : 3},               # 设定 num_friends = 3
             lambda row: row['user_id'] == 1)   # 在user_id == 1的行中
```

23.3 DELETE

SQL 有两种方法可以在表中删除行。一个比较有风险的方法是直接删掉表的每行：

```
DELETE FROM users;
```

稍微安全一点的方法是增加 WHERE 子句，再删掉匹配某种条件的行：

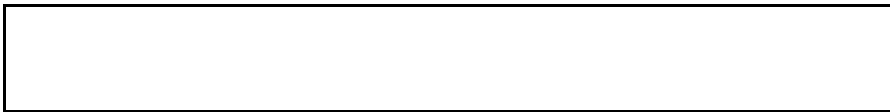
```
DELETE FROM users WHERE user_id = 1;
```

在表中增加这个功能很容易：

```
    def delete(self, predicate=lambda row: True):
        """delete all rows matching predicate
        or all rows if no predicate supplied"""
        self.rows = [row for row in self.rows if not(predicate(row))]
```

如果你提供了一个 predicate 函数（即 WHERE 子句），它会仅删掉那些满足条件的行。如果你不提供任何 predicate 函数，默认的判定就返回真值 True，然后删掉每一行。

例如：



```
users.delete(lambda row: row["user_id"] == 1)   # 删掉user_id == 1的行
users.delete()                                  # 删掉每一行
```

23.4 SELECT

通常，我们不会直接查看 SQL 表，而是通过一个 SELECT 语句查询：

```
SELECT * FROM users;                               -- 得到所有内容
SELECT * FROM users LIMIT 2;                       -- 得到前两行
SELECT user_id FROM users;                         -- 只得到特定列
SELECT user_id FROM users WHERE name = 'Dunn';     -- 只得到特定行
```

你也可以使用 SELECT 语句计算字段：

```
SELECT LENGTH(name) AS name_length FROM users;
```

我们会给 Table 类一个 select() 方法来返回一个新 Table。这个方法采用两种可选语句。

- keep_columns 声明了你希望在结果中保留的列名。如果没提供这一项，结果会包含所有的列。
- additional_columns 是一个字典，它的键是新列名，值是指定如何计算新列值的函数。

如果两种都不用，你只会得到一个表的机械复制的版本：

```
def select(self, keep_columns=None, additional_columns=None):

    if keep_columns is None:          # 如果没有指定列
        keep_columns = self.columns   # 则返回所有列

    if additional_columns is None:
        additional_columns = {}

    # 结果的新表
    result_table = Table(keep_columns + additional_columns.keys())

    for row in self.rows:
        new_row = [row[column] for column in keep_columns]
        for column_name, calculation in additional_columns.iteritems():
```

```
            new_row.append(calculation(row))
        result_table.insert(new_row)

    return result_table
```

select() 会返回新的表，而一般的 SQL SELECT 仅会生成某种临时结果集（除非你显式地将结果导入一个表）。

我们也需要 where() 和 limit() 方法。两种方法都很简单：

```
def where(self, predicate=lambda row: True):
    """return only the rows that satisfy the supplied predicate"""
    where_table = Table(self.columns)
    where_table.rows = filter(predicate, self.rows)
    return where_table

def limit(self, num_rows):
    """return only the first num_rows rows"""
    limit_table = Table(self.columns)
    limit_table.rows = self.rows[:num_rows]
    return limit_table
```

然后我们可以轻松创建与先前的 SQL 语句等价的 NotQuiteABase 语句：

```
# SELECT * FROM users;
users.select()

# SELECT * FROM users LIMIT 2;
users.limit(2)

# SELECT user_id FROM users;
users.select(keep_columns=["user_id"])

# SELECT user_id FROM users WHERE name = 'Dunn';
users.where(lambda row: row["name"] == "Dunn") \
     .select(keep_columns=["user_id"])

# SELECT LENGTH(name) AS name_length FROM users;
def name_length(row): return len(row["name"])

users.select(keep_columns=[],
             additional_columns = { "name_length" : name_length })
```

注意——不同于本书其他部分——我在这里用反斜线 \ 来表示程序语句的跨行延续。与其他方法相比，我认为这种方法可以让串连在一

起的 NotQuiteABase 查询语句更易读。

23.5 GROUP BY

另一种常见的 SQL 操作是 GROUP BY，它可以将在特定列有相同值的行进行分组，并求出特定的汇总值，如 MIN、MAX、COUNT 或 SUM。（回想 10.3 节“处理数据”中提到的 group_by 函数。）

例如，你需要对每个可能的名字长度找出相应的用户数目和最小 user_id：

```
SELECT LENGTH(name) as name_length,
 MIN(user_id) AS min_user_id,
 COUNT(*) AS num_users
FROM users
GROUP BY LENGTH(name);
```

我们通过 SELECT 生成的每个字段，要么需要在 GROUP BY 语句中完成（name_length 就是这样），要么需要汇总计算（min_user_id 和 num_users 就是这样）。

SQL 同样支持 HAVING 子句，它和 WHERE 子句类似，只是前者只对汇总结果过滤（而后者在汇总计算之前就过滤）。

也许你想知道名字以某些特定字母开头的用户的平均朋友数，结果却只看到了平均个数大于 1 的结果。（是的，其中某些例子是人为的。）

```
SELECT SUBSTR(name, 1, 1) AS first_letter,
 AVG(num_friends) AS avg_num_friends
FROM users
GROUP BY SUBSTR(name, 1, 1)
HAVING AVG(num_friends) > 1;
```

（处理字符串的函数会基于不同的 SQL 实现而有所不同；一些数据库会用 SUBSTRING 或者别的方式。）

你也可以计算总体的汇总值，这时我们不用 GROUP BY：

```
SELECT SUM(user_id) as user_id_sum
FROM users
```

```
WHERE user_id > 1;
```

为了对 NotQuiteABase 表增加这项函数功能，我们增加一个 group_by() 方法。它首先取你希望分组的列名、你希望对每组运行的汇总函数的字典和作用于多行的可选判定函数 having。

然后完成以下步骤。

1. 生成默认字典，将（按值分组的）元组映射到行（包含按值分的组）。前面提过，列表不可以用作字典的键；你得使用元组。

2. 遍历表的每一行，填充 defaultdict。

3. 用正确的输出列生成新表。

4. 遍历 defaultdict，并填充输出表。使用 having 过滤（如果有的话）。

（实际的数据库会用更有效的方式完成这些步骤。）

```
def group_by(self, group_by_columns, aggregates, having=None):

    grouped_rows = defaultdict(list)

    # 填充组
    for row in self.rows:
        key = tuple(row[column] for column in group_by_columns)
        grouped_rows[key].append(row)

    # 结果表中包含组列与汇总
    result_table = Table(group_by_columns + aggregates.keys())

    for key, rows in grouped_rows.iteritems():
        if having is None or having(rows):
            new_row = list(key)
            for aggregate_name, aggregate_fn in aggregates.iteritems():
                new_row.append(aggregate_fn(rows))
            result_table.insert(new_row)

    return result_table
```

我们再来看看为了完成先前 SQL 语句的功能还能用什么别的方法。name_length 标准表示如下所示：

```
def min_user_id(rows): return min(row["user_id"] for row in rows)
```

```
stats_by_length = users \
    .select(additional_columns={"name_length" : name_length}) \
    .group_by(group_by_columns=["name_length"],
              aggregates={ "min_user_id" : min_user_id,
                           "num_users" : len })
```

first_letter 的标准表示是：

```
def first_letter_of_name(row):
    return row["name"][0] if row["name"] else ""

def average_num_friends(rows):
    return sum(row["num_friends"] for row in rows) / len(rows)

def enough_friends(rows):
    return average_num_friends(rows) > 1

avg_friends_by_letter = users \
    .select(additional_columns={'first_letter' : first_letter_of_name}) \
    .group_by(group_by_columns=['first_letter'],
              aggregates={ "avg_num_friends" : average_num_friends },
              having=enough_friends)
```

user_id_sum 的标准表示是：

```
def sum_user_ids(rows): return sum(row["user_id"] for row in rows)

user_id_sum = users \
    .where(lambda row: row["user_id"] > 1) \
    .group_by(group_by_columns=[],
              aggregates={ "user_id_sum" : sum_user_ids })
```

23.6 ORDER BY

你会经常需要对结果排序。比如，你也许想知道你前两个用户的名字（按字母顺序排序的）：

```
SELECT * FROM users
ORDER BY name
LIMIT 2;
```

可以通过给表提供一个具有排序（order）功能的 order_by() 方法来轻易实现：

```
def order_by(self, order):
    new_table = self.select()       # 进行一次复制
    new_table.rows.sort(key=order)
    return new_table
```

然后我们可以这样做：

```
friendliest_letters = avg_friends_by_letter \
    .order_by(lambda row: -row["avg_num_friends"]) \
    .limit(4)
```

SQL 中的 `ORDER BY` 可以让你为每个字段设定 `ASC`（升序排列）或 `DESC`（降序排列）；这里我们不得不把它并入到 `order` 函数中。

237 JOIN

关系型数据库的表通常是正则化的，意味着依照冗余最小化的原则进行组织。例如，当我们处理用户对 Python 的兴趣时，我们只能为每个用户提供一个包含他的兴趣的列表。

SQL 表并不会包含真正的列表，它的实现机制是生成第二个表 `user_interests`，这个表包含了从 `user_id` 到兴趣的一对多的关系。在 SQL 中，你可以这样做：

```
CREATE TABLE user_interests (
    user_id INT NOT NULL,
    interest VARCHAR(100) NOT NULL
);
```

而在 NotQuiteABase 中，你需要建立表：

```
user_interests = Table(["user_id", "interest"])
user_interests.insert([0, "SQL"])
user_interests.insert([0, "NoSQL"])
user_interests.insert([2, "SQL"])
user_interests.insert([2, "MySQL"])
```

 这样仍然有很多的冗余性——兴趣“SQL”存储在两个不同的地方。在实际数据库中，你需要将 `user\_id` 和 `interest\_id` 存在表 `user`

_interests 中，并建立将 interest_id 映射到 interest 的第 3 个表 interests，这样你就可以将每个兴趣名字仅仅存储一次。这里，我们会举比实际需要更复杂的例子。

当我们的数据跨表存储时，该如果分析？当然是把表 JOIN 在一起。JOIN 可以将左表的行和右表对应的行结合在一起，其中，如何“对应”取决于我们对 join 的具体设定。

例如，为了找出对 SQL 感兴趣的用户，你需要：

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

JOIN 意味着对 users 中的每行都要找到 user_id，并找出 user_interests 中包含相同 user_id 的每一行，把它们联系起来。

注意，我们得指定哪个表需要参与 JOIN，哪些列需要被 join ON。这是一种 INNER JOIN，返回的是条件匹配行（仅仅是行的组合）的组合。

LEFT JOIN 不仅返回匹配行的组合，也返回左表中无匹配行的行（这种情形下，来自右表的字段值全部为 NULL）。

通过 LEFT JOIN 可以很容易计算出每个用户的兴趣数目：

```
SELECT users.id, COUNT(user_interests.interest) AS num_interests
FROM users
LEFT JOIN user_interests
ON users.user_id = user_interests.user_id
```

LEFT JOIN 保证没任何兴趣的用户在并集结果数据集中仍有一席之地（来自表 user_interest 的字段值为 NULL），并且 COUNT 只对非 NULL 值进行计数。

NotQuiteABase 的 join() 实现更为严格一些——它只对两表有共同列的部分做合并。虽然如此，也不妨把它写出来：

```
def join(self, other_table, left_join=False):

    join_on_columns = [c for c in self.columns           # 两个表的列
                       if c in other_table.columns]

    additional_columns = [c for c in other_table.columns  # 右表中的列
                          if c not in join_on_columns]

    # 左表中所有列 + 右表增加的列
    join_table = Table(self.columns + additional_columns)

    for row in self.rows:
        def is_join(other_row):
            return all(other_row[c] == row[c] for c in join_on_columns)

        other_rows = other_table.where(is_join).rows

        # 每对匹配的行生成一个新行
        for other_row in other_rows:
            join_table.insert([row[c] for c in self.columns] +
                              [other_row[c] for c in additional_columns])

        # 如果没有行匹配，在左并集的操作下生成空值
        if left_join and not other_rows:
            join_table.insert([row[c] for c in self.columns] +
                              [None for c in additional_columns])

    return join_table
```

这样我们就找到了对 SQL 感兴趣的用户：

```
sql_users = users \
    .join(user_interests) \
    .where(lambda row: row["interest"] == "SQL") \
    .select(keep_columns=["name"])
```

并且可以获得兴趣的数目：

```
def count_interests(rows):
    """counts how many rows have non-None interests"""
    return len([row for row in rows if row["interest"] is not None])

user_interest_counts = users \
    .join(user_interests, left_join=True) \
    .group_by(group_by_columns=["user_id"],
              aggregates={"num_interests" : count_interests })
```

在 SQL 中，还有一个 RIGHT JOIN，它查询的是右表中没有匹配的行。还有 FULL OUTER JOIN，它查询的是两表中无匹配的所有行。我们不再一一展示。

23.8 子查询

在 SQL 中，你可以把查询结果当成表，执行 SELECT（或 JOIN）操作。因此，如果你希望找到对 SQL 感兴趣的用户中的最小 user_id，可以考虑使用子查询。（当然，你也可以通过 JOIN 来做相同的运算，只是这样就无法展示子查询了。）

```
SELECT MIN(user_id) AS min_user_id FROM
(SELECT user_id FROM user_interests WHERE interest = 'SQL') sql_interests;
```

如果设计好了 NotQuiteABase，我们就可以方便地得到如下结果。（我们的查询结果是实际的表。）

```
likes_sql_user_ids = user_interests \
    .where(lambda row: row["interest"] == "SQL") \
    .select(keep_columns=['user_id'])

likes_sql_user_ids.group_by(group_by_columns=[],
                            aggregates={ "min_user_id" : min_user_id })
```

23.9 索引

为了找出包含特定值（比如名字是“Hero”）的行，NotQuiteABase 需要检查表的每一行。如果表有很多行，得花费很长时间。

同样，我们的 join 算法也极端低效。为了确认一个匹配，对左表的每一行，都需要检查右表的每一行。如果对象是两个大规模的表，这样的计算几乎没有结束的时候。

并且，你有时会需要对某些列加以约束。比如，在表 user 中，你很可能不希望两个不同的用户共用一个 user_id。

索引可以解决以上问题。如果表 user_interests 有一个基于 user_id 的索引，那么使用一个智能的 join 算法就可以不用遍历全表而直接完成匹配。如果表 users 已有基于 user_id 的“唯

一”索引，你再插入重复记录时会报错。

数据库中的每个表都有一个或多个索引，这让你可以通过关键词列快速查找行，可以有效并表，以及可以对行或者行集合施加唯一约束。

设计好、用好索引从某种程度上来讲更像魔术（还会因具体数据库的不同而情况各异），但如果你需要处理大量数据库工作，学习一下索引还是值得的。

23.10　查询优化

回想一下找出所有对 SQL 感兴趣的用户的查询方法：

```
SELECT users.name
FROM users
JOIN user_interests
ON users.user_id = user_interests.user_id
WHERE user_interests.interest = 'SQL'
```

在 NotQuiteABase 中，有（至少）两种不同的方法可以写这个查询。你可以在并表之前过滤表 `user_interests`：

```
user_interests \
    .where(lambda row: row["interest"] == "SQL") \
    .join(users) \
    .select(["name"])
```

或者你可以对并表的结果过滤：

```
user_interests \
    .join(users) \
    .where(lambda row: row["interest"] == "SQL") \
    .select(["name"])
```

两种方法都可以获得相同的结果，但先过滤再并表的方式效率更高，因为这样可以在并表时操作更少的行。

在 SQL 中，你基本不用担心这个。你只需“声明”自己需要的结果，再把任务丢给查询引擎（和有效使用索引）来实现即可。

23.11 NoSQL

数据库的一个近期发展趋势是非关系型的“NoSQL”数据库，这种数据库不将数据存放在表中。例如 MongoDB 这种流行的无结构数据库，它的元素直接是一些复杂 JSON 文档，而不是行。

此外，还有以列而非行的形式存储数据的列型数据库（适用于数据本身有很多列但查询却很少用到的情形），优化的通过键来检索单独（复杂）的值的键值存储对数据库，用来存储和遍历图像的数据库，跨多个数据中心运行的优化数据库，在内存中运行的数据库，以及存储时间序列数据的数据库等上百个数据库。

未来瞬息万变，谁也无法预知明天会流行什么，所以我仅仅告诉你有 NoSQL 这么一回事。因此你现在知道有 NoSQL 这个东西就行了。

23.12 延伸学习

- 如果你想下载关系型数据库的相关练习，SQLite（http://www.sqlite.org/）是个很好的选择，它快捷而精致。MySQL（http://www.mysql.com/）和 PostgreSQL（http://www.postgresql.org/）则规模更大，功能更多。这些软件都是免费的，而且有大量文档可供参考。

- 如果你想深入学习 NoSQL，我推荐 MongoDB（https://www.mongodb.org/），它上手特别容易，不过这个特点既受赞扬又被诟病。这个软件也拥有非常友好的文档。

- 维基百科上关于 NoSQL 的文章（https://en.wikipedia.org/wiki/NoSQL）非常全面，现在其所提供的链接涉及的有些数据库在撰写本书时甚至还没诞生。

第 24 章　MapReduce

明天已经照耀现在，只是尚未洒满每个角落。

——威廉 · 吉布森

MapReduce 是一个用来在在大型数据集上执行并行处理的算法模型。尽管这是一个非常强大的技术，但它的基本原理却很简单。

假设我们有一组待处理的项目，这些项目可能是网页日志、许多本书的文本、图像文件或者是其他东西。一个基本的 MapReduce 算法包括下面几个步骤。

1. 使用 `mapper` 函数把每个项目转化成零个或多个键值对。（通常这被称为 `map` 函数，但是已经有了一个叫 `map` 的 Python 函数，所以我们不能把它们混淆。）

2. 用相同的键把所有的键值对收集起来。

3. 在每一个分好组的值集合上使用 `reducer` 函数，对每个对应的键生成输出值。

这样讲是很抽象的，所以让我们看一个具体的例子。数据科学里很少有绝对的规则，但有一个不成文的规则是，你面对的第一个 MapReduce 的例子中必须要涉及单词计数。

24.1　案例：单词计数

DataSciencester 网站的用户数量已增长到了数百万！这对你的工作是极大的保障，但也使日常分析变得有点困难了。

比如，内容部门的副总想知道用户的状态更新都涉及什么内容。作为初步的尝试，你决定统计一下出现的单词的数量，这样你就可以准备一个关于出现频度最高的单词的报告。

当有几百个用户的时候这做起来很简单：

```
def word_count_old(documents):
    """word count not using MapReduce"""
```

```
    return Counter(word
        for document in documents
        for word in tokenize(document))
```

而数百万个用户的 documents 集合就变得很大，你的电脑都装不下。如果你能把它们纳入 MapReduce 模型处理，就可以使用你的引擎上已经部署的一些“大数据”架构。

首先，我们需要一个函数来把文档转化成一系列的键值对。我们希望输出的结果能按单词分组，这意味着键应该是单词。对每一个单词，我们只发送值 1 来表示这个键值对对应于单词出现一次：

```
def wc_mapper(document):
    """for each word in the document, emit (word,1)"""
    for word in tokenize(document):
        yield (word, 1)
```

先暂时跳过第二步，假设对某些词我们已经收集了所发送过的对应计数的列表。接下来生成我们所需要的这个词的全部计数：

```
def wc_reducer(word, counts):
    """sum up the counts for a word"""
    yield (word, sum(counts))
```

再返回步骤 2，现在我们需要收集来自 wc_mapper 的结果，再把它们传递给 wc_reducer。让我们思考一下怎么在一台计算机上完成这件事：

```
def word_count(documents):
    """count the words in the input documents using MapReduce"""

    # 存放分好组的值
    collector = defaultdict(list)

    for document in documents:
        for word, count in wc_mapper(document):
            collector[word].append(count)

    return [output
            for word, counts in collector.iteritems()
            for output in wc_reducer(word, counts)]
```

假设我们有三个文档 ["data science", "big data", "science fiction"]。

然后把 wc_mapper 应用到第一个文档，产生两个键值对 ("data", 1) 和 ("science", 1)。在处理完所有三个文档之后，列表 collector 会包含：

```
{ "data" : [1, 1],
  "science" : [1, 1],
  "big" : [1],
  "fiction" : [1] }
```

然后 wc_reducer 函数生成了每个单词的计数：

```
[("data", 2), ("science", 2), ("big", 1), ("fiction", 1)]
```

24.2 为什么是MapReduce

如同早先所提到的，MapReduce 的主要优点就是通过将处理过程移动到数据来进行分布式的计算。假设我们想对数以十亿计的文档进行单词计数。

我们最初的（非 MapReduce 的）方法要求机器在每一个文档上进行处理。这意味着所有的文档要么存在机器上要么在处理期间转移到机器上。更重要的是，这意味着机器一次只能处理一个文档。



如果机器是多核的，且如果代码是为了利用多核的优势而重写过的，那它是有可能一次处理几个文档的。但即使这样，所有的文档也都要放到机器中来。

假设现在我们将数十亿个文档分散到 100 台机器上。利用正确的架构（并且掩盖掉一些细节），我们可以做下面的事。

- 让每一台机器在它的文档上运行 mapper 函数，产生大量的键值对。
- 把那些键值对分配到一些“reducing”的机器上，确保对应任何一个给定键的对在同一台机器上完成计算。
- 每一台 reducing 机器通过键分组这些对，然后对每个值的集合

运行 reducer 函数。

- 返回每个键值对。

它可以处理的横向规模水平令人惊叹。如果我们把机器的数量加倍，那么（忽略运行 MapReduce 系统的某些固定成本）计算的运行速度大约会快两倍。每一个 mapper 机器只需要做一半的工作，（假设有足够多不同的键来进一步分配 reducer 工作）reducer 机器也是。

24.3 更加一般化的MapReduce

思考一下，前面例子中所有的单词计数代码是包括在 wc_mapper 和 wc_reducer 这两个函数中的。这意味着通过几个改变我们会得到一个更通用的框架（仍然是运行在单个机器上的）：

```
def map_reduce(inputs, mapper, reducer):
    """runs MapReduce on the inputs using mapper and reducer"""
    collector = defaultdict(list)

    for input in inputs:
        for key, value in mapper(input):
            collector[key].append(value)

    return [output
            for key, values in collector.iteritems()
            for output in reducer(key,values)]
```

然后我们可以通过下面的方法简单地完成单词计数：

```
word_counts = map_reduce(documents, wc_mapper, wc_reducer)
```

这为我们解决各种类型的问题提供了灵活性。

在继续讲解下面的内容之前，我们先观察一下 wc_reducer 函数，它仅仅是把对应于每一个键的值加起来。这种聚合是非常普遍的，值得把它抽象出来：

```
def reduce_values_using(aggregation_fn, key, values):
    """reduces a key-values pair by applying aggregation_fn to the values'
    yield (key, aggregation_fn(values))

def values_reducer(aggregation_fn):
```

```
    """turns a function (values -> output) into a reducer
    that maps (key, values) -> (key, output)"""
    return partial(reduce_values_using, aggregation_fn)
```

在这之后我们就可以轻松创建以下内容：

```
sum_reducer = values_reducer(sum)
max_reducer = values_reducer(max)
min_reducer = values_reducer(min)
count_distinct_reducer = values_reducer(lambda values: len(set(values)))
```

24.4　案例：分析状态更新

内容部门的副总对单词计数印象深刻，并询问你还能从用户的状态更新中学到什么。你设法提取了一个类似下面这样的状态更新的数据集：

```
{"id": 1,
 "username" : "joelgrus",
 "text" : "Is anyone interested in a data science book?",
 "created_at" : datetime.datetime(2013, 12, 21, 11, 47, 0),
 "liked_by" : ["data_guy", "data_gal", "mike"] }
```

假设我们想找出每周内的哪一天人们讨论数据科学最多。为了找到这个结果，只需计算一下一周内每一天有多少个数据科学更新。这意味着我们需要按每周内的每天进行分组，这就是我们的键。而且如果对每一个包含“数据科学”的更新发送一个值 `1`，就可以使用 `sum` 函数得到总数：

```
def data_science_day_mapper(status_update):
    """yields (day_of_week, 1) if status_update contains "data science" ""
    if "data science" in status_update["text"].lower():
        day_of_week = status_update["created_at"].weekday()
        yield (day_of_week, 1)

data_science_days = map_reduce(status_updates,
                               data_science_day_mapper,
                               sum_reducer)
```

再举一个稍微复杂一点的例子，假设我们需要找出每个用户在其状态更新中最常用的单词是什么。为了建立 `mapper`，我们的脑海中会浮

现以下三种方法。

- 把用户名放到键当中；把单词和计数放到值当中。
- 把单词放到键当中；把用户名和计数放到值当中。
- 把用户名和单词放到键当中；把计数放到值当中。

稍作考虑，我们很容易就能判断出应该按 `username` 分组，因为我们想分别考虑每个人的单词。而且我们不想用 `word` 来分组，因为我们的 reducer 需要看到每个人所有的词汇，以此找到哪一个是最流行的。这意味着第一种选择是最优选择：

```
def words_per_user_mapper(status_update):
    user = status_update["username"]
    for word in tokenize(status_update["text"]):
        yield (user, (word, 1))

def most_popular_word_reducer(user, words_and_counts):
    """given a sequence of (word, count) pairs,
    return the word with the highest total count"""

    word_counts = Counter()
    for word, count in words_and_counts:
        word_counts[word] += count

    word, count = word_counts.most_common(1)[0]

    yield (user, (word, count))

user_words = map_reduce(status_updates,
                        words_per_user_mapper,
                        most_popular_word_reducer)
```

或者我们能为每个用户找到各自的状态点赞者的个数：

```
def liker_mapper(status_update):
    user = status_update["username"]
    for liker in status_update["liked_by"]:
        yield (user, liker)

distinct_likers_per_user = map_reduce(status_updates,
                                      liker_mapper,
                                      count_distinct_reducer)
```

24.5 案例：矩阵计算

回想 22.2.1 节“矩阵乘法”，给定一个 $m \times n$ 的矩阵 A 和一个 $n \times k$ 的矩阵 B，可以把它们乘起来得到 $m \times k$ 的矩阵 C，其中 C 的第 i 行第 j 列的元素由下式给出：

$$C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \cdots + A_{in}B_{nj}$$

如同我们之前所见的，表示一个 $m \times n$ 矩阵的“自然的”方法是列表的列表，其中元素 A_{ij} 是第 i 个列表的第 j 个元素。

但大型矩阵有时候是稀疏的，即大部分的元素等于 0。对于大型稀疏矩阵而言，列表的列表是一种非常浪费的表达方式。一种更简洁的表达方式是元组的列表 `(name, i, j, value)`，其中 `name` 代表矩阵，而 `i`、`j`、`value` 表示一个非零元素的位置。

比如，一个十亿 × 十亿的矩阵会有亿亿级别（quintillion，1×10^{18}）的元素，这是难以存储在一个计算机中的。但是如果每行当中只有不多的一些非零元素，上面那种替代的表示法就会小很多个数量级。

基于这种表示法，我们可以使用 MapReduce 以分布式的方式执行矩阵乘法。

为使用这种算法，请注意，A_{ij} 只用于计算 C 的第 i 行的元素，B_{ij} 只用于计算 C 的第 j 列的元素。我们的目标是使 `reducer` 的每一个输出构成矩阵 C 的一个元素。这意味着我们需要用 `mapper` 发送键值，以确定 C 中的每个元素。建议像下面这样处理：

```
def matrix_multiply_mapper(m, element):
    """m is the common dimension (columns of A, rows of B)
    element is a tuple (matrix_name, i, j, value)"""
    name, i, j, value = element

    if name == "A":
        # A_ij是每个C_ik之和的第j个元素，其中k=1..m
        for k in range(m):
            # 与C_ik的其他元素分组
            yield((i, k), (j, value))
    else:
        # B_ij是每个C_kj之和的第i个元素
        for k in range(m):
            # 与C_kj的其他元素分组
            yield((k, j), (i, value))
```

```
def matrix_multiply_reducer(m, key, indexed_values):
    results_by_index = defaultdict(list)
    for index, value in indexed_values:
        results_by_index[index].append(value)

    # 对有两个结果的位置把所有的乘积加起来
    sum_product = sum(results[0] * results[1]
                      for results in results_by_index.values()
                      if len(results) == 2)

    if sum_product != 0.0:
        yield (key, sum_product)
```

比如，如果你有如下的两个矩阵：

```
A = [[3, 2, 0],
     [0, 0, 0]]

B = [[4, -1, 0],
     [10, 0, 0],
     [0, 0, 0]]
```

你可以把它们重写为元组：

```
entries = [("A", 0, 0, 3), ("A", 0, 1,  2),
           ("B", 0, 0, 4), ("B", 0, 1, -1), ("B", 1, 0, 10)]
mapper = partial(matrix_multiply_mapper, 3)
reducer = partial(matrix_multiply_reducer, 3)

map_reduce(entries, mapper, reducer) # [((0, 1), -3), ((0, 0), 32)]
```

在这样一个小矩阵上操作并不太有趣，但是如果你有百万行百万列的矩阵，MapReduce 就会起很大作用。

24.6　题外话：组合器

你很可能已经注意到，许多 mapper 可能包括一些额外的信息。比如，在计数单词的时候，与其发送 (word, 1) 并累加求和，不如发送 (word, None) 再取其长度。

我们没有这么做的一个原因是，在分布式情景下，我们有时会想用组合器（combiner）来缩减在计算机之间转移的数据数量。如果一个 mapper 机器发现单词“data”500 次，可以让它在移交数据给缩减机

器之前把 500 个 `("data", 1)` 组合成一个单独的 `("data", 500)`。这使得转移的数据少了很多，算法会大大加快。

基于我们编写 reducer 的方法，它能正确地处理这些组合的数据。（如果我们已经用 `len` 函数写了 reducer，它就不能处理组合了。）

24.7 延伸学习

- 使用最为广泛的 MapReduce 系统是 Hadoop（http://hadoop.apache.org/），它值得用很多本书来阐述。它有许多商业或非商业的发行版，以及一个由 Hadoop 相关工具组成的巨大的生态系统。

  为了使用 Hadoop，你需要建立自己的聚类（或者可以在允许的情况下使用别人建好的聚类），当然，对于承压能力较差的人来说，可以不把这当成必然的任务。Hadoop mapper 和 reducer 通常是用 Java 写成的，尽管有一种被称为“Hadoop streaming”的功能允许你用其他的语言（包括 Python）来写。

- Amazon.com 提供 Elastic MapReduce（http://aws.amazon.com/cn/elasticmapreduce/）服务，它可以用编程的方式来创建或销毁聚类，并只根据你使用该服务的时长来收费。

- mrjob（https://github.com/Yelp/mrjob）是 Python 的一个 Hadoop（或 Elastic MapReduce）的接口包。

- Hadoop 任务是典型的高延迟的，这对于“实时分析”来说不是个好选择。有多种建立在 Hadoop 之上的“实时分析”工具，同时还有一些替代性的框架也日益流行。最流行的两种是 Spark（http://spark.apache.org/）和 Storm（http://storm.incubator.apache.org/）。

- 总之，很有可能目前流行的某种新的分布式框架在本书写作时还未问世，那就需要你自己把它找出来了。

第 25 章　数据科学前瞻

此刻，我再次祈求我丑陋的后代生生世世繁荣昌盛。

——玛丽 · 雪莱

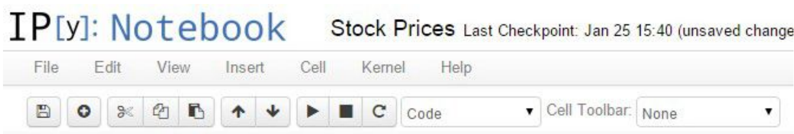
从这里出发，要去哪里呢？如果至此讲的有关数据科学的东西还没有把你吓跑的话，接下来你可以学习以下内容。

25.1　IPython

在本书前面的章节我们提到过 IPython（http://ipython.org/）。它提供了一个远比标准 Python shell 功能强大的 shell，而且加入了一些“魔法函数”，可以让你（比如）轻松地复制粘贴代码（以空行和空白的组合形式实现代码通常是比较复杂的），而且可以在 shell 内部运行代码。

掌握了 IPython 会让你的工作变得非常轻松。（甚至仅仅学一点点 IPython 就可以了。）

此外，它还允许你创建将文本、Python 动态代码和可视化相结合的“日记本”（notebook），你可以将它们与别人共享，或仅仅保存为自己的日志（图 25-1）。



Code ▾ Cell Toolbar: None ▾

```
In [1]: import csv
```

Here's where we read from the file:

```
In [2]: with open(r"c:\src\data-science-from-scratch\code\stocks.txt", "rb") as f:
            reader = csv.DictReader(f, delimiter='\t')
            data = [row for row in reader]
```

What does this data look like?

```
In [3]: print data[0]
        {'date': '2015-01-23', 'symbol': 'AAPL', 'closing_price': '112.98'}
```

Now we can find the maximum price for AAPL stock using a list comprehension:

```
In [4]: print max(row["closing_price"] for row in data if row["symbol"] == "AAPL")
        99.68
```

图 25-1：一个 **IPython** 日记本

25.2　数学

本书涵盖了线性代数（第 4 章）、统计（第 5 章）、概率（第 6 章）和机器学习的一些内容。

要想成为一名优秀的科学家，你需要知道更多关于这些领域的知识，而且我鼓励你对每一个领域开展深入的学习。你可以参考我在每一章末尾推荐的教科书，也可以使用自己选择的教科书，或通过在线课程甚至线下课程来进行学习。

25.3　不从零开始

“从零开始”做一件事情对于理解这件事情的工作原理是很有好处的。但这种方法对于性能表现（除非你仅仅是出于性能方面的考虑而实现它们）、易用性、快速成型或误差处理来说并不是很理想。

在实践中，你可能需要用到精心设计的能稳定地实施基础原则的库。（我本来是想在本书中加一个“让我们来学一些库”的板块，幸好，被 O'Reilly 否决了。）

25.3.1　NumPy

NumPy（即“Numeric Python”，http://www.numpy.org/）提供了处理“真实”科学计算的工具。它提供了比我们的 `list` 向量性能更好的数组，提供了比我们的 `list-of-list` 矩阵性能更好的矩阵，以及大量利用它们来工作的数值函数。

对很多其他的库来说，NumPy 是个基础构件，这是它特别值得学习的原因。

25.3.2　pandas

pandas（http://pandas.pydata.org/）提供了处理 Python 数据集的更多的数据结构。它主要的抽象概念是 DataFrame，在内容上与我们在第 23 章中构建的 NotQuiteABase `table` 类很相似，但是有更多的功能和更好的性能。如果你打算用 Python 修改、分划、分组或操作数据集，pandas 是一个非常有用的工具。

25.3.3　scikit-learn

scikit-learn（http://scikit-learn.org/）可能是 Python 中处理机器学习问题最常用的库。它包括我们用过的所有模型以及很多我们没用过的模型。在真实的问题上，你无需从零开始建立决策树，而是可以使用 scikit-learn 来做繁重的工作。在真实的问题上，你也无需手动写出优化算法，而是可以依靠 scikit-learn 使用已有的优秀算法。

scikit-learn 的文档包括了许许多多案例（http://scikit-learn.org/stable/auto_examples/）来说明它可以做什么（或者，更一般地，说明机器学习可以做什么）。

25.3.4　可视化

我们创建过的 `matplotlib` 图形很清晰而且功能强大，但它还不够美观（而且一点交互性也没有）。如果想深入了解数据可视化，你可以有多种选择。

第一种是深入学习 `matplotlib`，因为我们涉及的内容只是它的一小部分。在它的网站上有许多关于它的功能的例子（http://matplotlib.org/examples/），还有一些更有趣的可视化的图库（http://matplotlib.org/gallery.html）。如果你打算创建静态的可视化（比如想把它印在书里），这可能是你下一步最好的选择。

你也应该试试 seaborn（http://web.stanford.edu/~mwaskom/software/seaborn/），这是一个可以使 `matplotlib` 更有吸引力（还有其他很多优点）的库。

如果你想创建那种可以在网络上分享的交互式可视化，D3.js（http://d3js.org/）是首选，它是一个可用于创建“数据驱动文档”（data driven documents，名称中的“3D”便由此得来）的 JavaScript 库。即使你不太懂 JavaScript，通常也可以从 D3 的图库（https://github.com/mbostock/d3/wiki/Gallery）中找到可以模仿的例子，并套用到你的数据上。（好的数据科学家从 D3 的图库中复制例子，出色的数据科学家从 D3 的图库中“偷”例子。）

即使你对 D3 一点兴趣也没有，仅仅浏览一下它的图库也能学到不少关于数据可视化的东西。

Bokeh（http://bokeh.pydata.org/en/latest/）是一个把 D3 风格的功能整合到 Python 中的项目。

25.3.5 R

尽管你可以完全不用学习 R（http://www.r-project.org/），但许多数据科学家和数据科学项目都会用到 R，所以起码应该熟悉它。

这一部分是因为这样做有益于你理解别人基于 R 的博客、案例和代码，一部分是因为这样可以让你更好地领略 Python 的（相对的）清晰和优雅。还有一部分原因是，这会让你在永不停息的“R 好还是 Python 好”的口水战中成为一个更加见多识广的专家。

这个世界从不缺乏 R 的教程、R 的课程和 R 的图书。我听说 *Hands-On Programming with R*（http://shop.oreilly.com/product/0636920028574.do）这本书不错，不仅仅是因为它也是 O'Reilly 出版的书。（好吧，主要就是因为它是 O'Reilly 出版的书。）

25.4 寻找数据

如果你把从事数据科学作为你工作的一部分，那么你会很愿意把获得数据也作为你工作的一部分（尽管并不一定）。如果你把从事数据科学工作视为乐趣将会如何？数据是无处不在的，但下面这些资源可以是很好的起点。

- Data.gov（http://www.data.gov/）是政府开放数据的门户网站。如果你想找任何和政府有关的数据（现在看这可能涉及方方面面的事情），它是个很好的开始。

- reddit 上有 r/datasets（https://www.reddit.com/r/datasets）和 r/data（http://www.reddit.com/r/data）两个论坛，是一个可以请求数据和发现数据的地方。

- Amazon.com 上有一些公用数据集（http://aws.amazon.com/cn/public-data-sets/），它希望你能使用它的产品来分析这些数据集（但你可以用任何你想用的产品来分析）。

- Robb Seaton 的博客上有一个富有创意的专业数据集的列表（http://rs.io/100-interesting-data-sets-for-statistics/）。

- Kaggle（https://www.kaggle.com/）是一个举办数据科学竞赛的网站。我从没成功跻身这个比赛（在数据科学领域我没有多少竞争力），但没准你就会成功。

25.5　从事数据科学

翻阅数据目录固然不错，但是最好的项目（和产品）还是那些让人心动的。下面列举我做过的其中一些项目。

25.5.1　Hacker News

Hacker News（https://news.ycombinator.com/news）是一个聚集并讨论与技术相关的新闻的网站。它收集了大量文章，但很多对我来说并不有趣。

因此，数年之前，我着手建立了一个 Hacker News 的内容分类器（https://github.com/joelgrus/hackernews）来预测我是否喜欢某篇给定的文章。这一做法可能会受到有些 Hacker News 用户的排斥，他们会抱怨怎么还会有人不喜欢 Hacker News 的所有文章。

这项工作涉及对大量文章的手动标注（为了建立训练集），选择文章的特征（比如标题中的单词和链接的域），以及训练一个和之前我们做过的垃圾邮件检测类似的朴素贝叶斯分类器。

不知道出于什么心理，我当时是用 Ruby 建立的分类器。吸取我的教训吧。

25.5.2 消防车

我生活在西雅图市中心的一条主要街道上，这条街是从一个消防站去往城市里大多数火灾（或者看起来疑似火灾）的必经之地。由此，多年来我产生了对西雅图消防部门的兴趣。

幸运的是（从数据的观点），消防部门运行着一个实时的 911 网站（http://www2.seattle.gov/fire/realtime911/getDatePubTab.asp），上面列出了每一次火警状况和出动的消防车。

所以，为了满足我的兴趣，我抓取了多年的火警数据并执行了关于消防车的社交网络分析（https://github.com/joelgrus/fire），这就要求我创造了一个特定的关于消防车的中心性的概念，我称之为 TruckRank。

25.5.3 T恤

我有一个小女儿，她的童年有一件令我感到十分沮丧的事情，就是“女孩的 T 恤”大都很单调乏味，而“男孩的 T 恤”却都充满趣味。

尤其是，我觉得出售给男童和女童的 T 恤之间有很明显的区别。所以我问自己能不能训练一个模型来识别这些区别。

剧透：我能（https://github.com/joelgrus/shirts）。

这项工作包括下载数百件 T 恤的图案，把它们改成同样的尺寸，再把它们转换为像素颜色的向量，最后使用逻辑回归建立分类器。

一种看起来较为简单的方法是针对每件 T 恤的颜色分类；第二种方法是找到 T 恤颜色向量的前 10 个主成分，然后把每件 T 恤投影到由 10 个“特征 T 恤”（eigenshirt）1 所组成的 10 维空间上进行分类（见图 25-2）。

> 1即 10 个主成分。——译者注



图 **25-2**：对应于第一个主成分的特征 **T** 恤

25.5.4　你呢？

什么事情会让你兴致勃发？什么问题会让你夜不能寐？去寻找相关的数据（或者抓取一些网站），对它们做一些数据科学的分析吧。

告诉我你的发现吧！通过 joelgrus@gmail.com 给我发邮件，或者去 Twitter @joelgrus 找我。

作者简介

Joel Grus 是 Google 的一位软件工程师，曾于数家创业公司担任数据科学家。目前住在西雅图，专注于数据科学工作并乐此不疲。偶尔在 joelgrus.com 发表博客，长期活跃于 Twitter @joelgrus。

关于封面

本书封面上的动物是岩雷鸟（学名 Lagopus muta），是松鸡家族中一种中等体型的猎鸟，在英国和加拿大叫“雷鸟”，在美国叫“雪鸡”。岩雷鸟喜定居，栖息于北极和靠近北极的欧亚大陆，以及北美的格陵兰岛地区，栖息地多为贫瘠而孤立之地，如苏格兰山脉、比利牛斯山、阿尔卑斯山、乌拉尔山脉、帕米尔高原、保加利亚、阿尔泰山脉和日本阿尔卑斯山脉。食物主要是桦树和柳树的芽，也包含种子、花、叶子、浆果等。发育中的雏鸟也吃昆虫。

雄性岩雷鸟并没有一般松鸡所具有的典型的羽饰，但它们有肉冠，可以帮助其求偶或与其他雄鸟争斗。大量研究证明，雄性岩雷鸟的肉冠尺寸与其睾酮水平之间有一定的相关性。岩雷鸟的羽毛冬季为白色，春夏会变为黑褐色，可起到季节性的保护作用。具有繁殖能力的雄鸟翅膀为白色，上半部分为灰色，但在冬季，除了尾羽呈黑色外，周身均为白色。

6 个月大的岩雷鸟即发育成熟，且通常每只雌鸟可孵化 6 只雏鸟，以帮助保护种群数量免受诸如狩猎等外界因素的影响。岩雷鸟主要被金雕捕食，而偏远孤立的栖息地帮助其躲过了很多其他捕食者。

岩雷鸟肉在冰岛是备受欢迎的节日餐主食。由于种群数量的下降，岩雷鸟的捕猎在 2003 年和 2004 年是被禁止的。2005 年，捕猎禁令解除，但特定时期内仍禁止捕猎。所有与岩雷鸟有关的交易都是非法的。

O'Reilly 封面上的动物很多都是濒临灭绝的，所有这些动物对世界来说都是很重要的。若想了解你力所能及的事，请访问 animals.oreilly.com。

封面图片来自 Cassell 的 *Natural History*。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_interview，讲述码农精彩人生
- 微信 图灵教育：turingbooks

091507240605ToBeReplacedWithUserId

版权信息

书名：Python数据挖掘入门与实践

作者：[澳] Robert Layton

译者：杜春晓

ISBN：978-7-115-42710-6

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

091507240605ToBeReplacedWithUserId

版权声明
译者序
前言
本书主要内容
本书的阅读前提
本书的目标读者
排版约定
读者反馈
客户支持
下载示例代码
下载配套PDF文件
勘误表
侵权
问题
第 1 章　开始数据挖掘之旅
1.1　数据挖掘简介
1.2　使用Python和IPython Notebook
1.2.1　安装Python
1.2.2　安装IPython
1.2.3　安装scikit-learn库
1.3　亲和性分析示例
1.3.1　什么是亲和性分析
1.3.2　商品推荐
1.3.3　在NumPy中加载数据集
1.3.4　实现简单的排序规则
1.3.5　排序找出最佳规则
1.4　分类问题的简单示例
1.5　什么是分类
1.5.1　准备数据集
1.5.2　实现OneR算法
1.5.3　测试算法
1.6　小结

第 2 章　用scikit-learn估计器分类
2.1　scikit-learn估计器
2.1.1　近邻算法
2.1.2　距离度量
2.1.3　加载数据集
2.1.4　努力实现流程标准化
2.1.5　运行算法
2.1.6　设置参数
2.2　流水线在预处理中的应用
2.2.1　预处理示例
2.2.2　标准预处理
2.2.3　组装起来
2.3　流水线
2.4　小结
第 3 章　用决策树预测获胜球队
3.1　加载数据集
3.1.1　采集数据
3.1.2　用pandas加载数据集
3.1.3　数据集清洗
3.1.4　提取新特征
3.2　决策树
3.2.1　决策树中的参数
3.2.2　使用决策树
3.3　NBA比赛结果预测
组装起来
3.4　随机森林
3.4.1　决策树的集成效果如何
3.4.2　随机森林算法的参数
3.4.3　使用随机森林算法
3.4.4　创建新特征
3.5　小结
第 4 章　用亲和性分析方法推荐电影

4.1 亲和性分析
4.1.1 亲和性分析算法
4.1.2 选择参数
4.2 电影推荐问题
4.2.1 获取数据集
4.2.2 用pandas加载数据
4.2.3 稀疏数据格式
4.3 Apriori算法的实现
4.3.1 Apriori算法
4.3.2 实现
4.4 抽取关联规则
评估
4.5 小结
第 5 章 用转换器抽取特征
5.1 特征抽取
5.1.1 在模型中表示事实
5.1.2 通用的特征创建模式
5.1.3 创建好的特征
5.2 特征选择
选择最佳特征
5.3 创建特征
主成分分析
5.4 创建自己的转换器
5.4.1 转换器API
5.4.2 实现细节
5.4.3 单元测试
5.4.4 组装起来
5.5 小结
第 6 章 使用朴素贝叶斯进行社会媒体挖掘
6.1 消歧
6.1.1 从社交网站下载数据
6.1.2 加载数据集并对其分类

6.1.3　Twitter数据集重建
6.2　文本转换器
6.2.1　词袋
6.2.2　N元语法
6.2.3　其他特征
6.3　朴素贝叶斯
6.3.1　贝叶斯定理
6.3.2　朴素贝叶斯算法
6.3.3　算法应用示例
6.4　应用
6.4.1　抽取特征
6.4.2　将字典转换为矩阵
6.4.3　训练朴素贝叶斯分类器
6.4.4　组装起来
6.4.5　用F1值评估
6.4.6　从模型中获取更多有用的特征
6.5　小结
第 7 章　用图挖掘找到感兴趣的人
7.1　加载数据集
7.1.1　用现有模型进行分类
7.1.2　获取Twitter好友信息
7.1.3　构建网络
7.1.4　创建图
7.1.5　创建用户相似度图
7.2　寻找子图
7.2.1　连通分支
7.2.2　优化参数选取准则
7.3　小结
第 8 章　用神经网络破解验证码
8.1　人工神经网络
神经网络简介
8.2　创建数据集

8.2.1 绘制验证码
8.2.2 将图像切分为单个的字母
8.2.3 创建训练集
8.2.4 根据抽取方法调整训练数据集
8.3 训练和分类
8.3.1 反向传播算法
8.3.2 预测单词
8.4 用词典提升正确率
8.4.1 寻找最相似的单词
8.4.2 组装起来
8.5 小结
第 9 章 作者归属问题
9.1 为作品找作者
9.1.1 相关应用和使用场景
9.1.2 作者归属
9.1.3 获取数据
9.2 功能词
9.2.1 统计功能词
9.2.2 用功能词进行分类
9.3 支持向量机
9.3.1 用SVM分类
9.3.2 内核
9.4 字符N元语法
抽取字符N元语法
9.5 使用安然公司数据集
9.5.1 获取安然数据集
9.5.2 创建数据集加载工具
9.5.3 组装起来
9.5.4 评估
9.6 小结
第 10 章 新闻语料分类
10.1 获取新闻文章

10.1.1 使用Web API获取数据
10.1.2 数据资源宝库reddit
10.1.3 获取数据
10.2 从任意网站抽取文本
10.2.1 寻找任意网站网页中的主要内容
10.2.2 组装起来
10.3 新闻语料聚类
10.3.1 k-means算法
10.3.2 评估结果
10.3.3 从簇中抽取主题信息
10.3.4 用聚类算法做转换器
10.4 聚类融合
10.4.1 证据累积
10.4.2 工作原理
10.4.3 实现
10.5 线上学习
10.5.1 线上学习简介
10.5.2 实现
10.6 小结
第 11 章 用深度学习方法为图像中的物体进行分类
11.1 物体分类
11.2 应用场景和目标
使用场景
11.3 深度神经网络
11.3.1 直观感受
11.3.2 实现
11.3.3 Theano简介
11.3.4 Lasagne简介
11.3.5 用nolearn实现神经网络
11.4 GPU优化
11.4.1 什么时候使用GPU进行计算
11.4.2 用GPU运行代码

11.5　环境搭建
11.6　应用
11.6.1　获取数据
11.6.2　创建神经网络
11.6.3　组装起来
11.7　小结
第 12 章　大数据处理
12.1　大数据
12.2　大数据应用场景和目标
12.3　MapReduce
12.3.1　直观理解
12.3.2　单词统计示例
12.3.3　Hadoop MapReduce
12.4　应用
12.4.1　获取数据
12.4.2　朴素贝叶斯预测
12.5　小结
附录　接下来的方向
第 1 章——开始数据挖掘之旅
Scikit-learn教程
扩展IPython Notebook
第 2 章——用scikit-learn估计器分类
k近邻算法的扩展
更多复杂的流水线
比较分类器
第 3 章——用决策树预测获胜球队
pandas的更多内容
更多复杂特征
第 4 章——用亲和性分析方法推荐电影
新数据集
Eclat算法
第 5 章——用转换器抽取特征

增加噪音
Vowpal Wabbit
第 6 章——使用朴素贝叶斯进行社会媒体挖掘3
垃圾信息监测
自然语言处理和词性标注
第 7 章——用图挖掘找到感兴趣的人
更复杂的算法
NetworkX
第 8 章——用神经网络破解验证码
好（坏？）验证码
深度网络
增强学习
第 9 章——作者归属问题
增加数据量
博客语料
局部N元语法
第 10 章——新闻语料分类
算法评价
近期趋势分析
实时聚类
第 11 章——用深度学习方法为图像中的物体进行分类
Keras和Pylearn2
Mahotas
第 12 章——大数据处理
Hadoop课程
Pydoop
推荐引擎
更多资源

版权声明

Copyright © 2015 Packt Publishing. First published in the English language under the title *Learning Data Mining with Python*.

Simplified Chinese-language edition copyright © 2016 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由Packt Publishing授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

译者序

云计算、大数据、物联网，这几年很火。到现在为止，人们对云计算的激情已经回落到比较理智的水平，各种云基础设施已投入使用，支撑起关系国计民生的信息化应用。物联网还在建设中，家电智能化、个人健康信息数字化、交通智能化等趋势在我们身边悄然推进。开放互联的概念不再囿于传统的互联网思维，我们生活所触及的一切正在被编织到一张包罗万象的大网之中。它将会对社会产生何种影响，我们拭目以待。虽然大数据现在很火，各种大数据研究中心相继建立，但这只是刚刚开始。随着更多的人和设备接入互联网，随着人们对世界认识的加深和新工具的研发，数据规模将加速膨胀，超乎想象。大数据的春天才刚刚到来。数据采集能力上去之后，势必要求数据挖掘能力跟得上。正如作者在第12章中讲到的，大数据带来的一个挑战就是，重要信息可能被垃圾信息湮没。由此我们不难推断出数据挖掘技术在发现、突显和传承人类文明方面将起到不可替代的作用。本书讲解的正是大数据时代的核心技能——数据挖掘技术，可以预见该项技术将发挥出越来越重要的作用。

本书讲解了如何用Python语言进行数据挖掘。Python是一种通用型编程语言，它简单易学，上手快，有着丰富的第三方库，社区氛围友好。从数据采集、分析一直到应用开发层面，Python都有成熟的库。使用Python语言进行开发，无需过多关注语言细节，开发者可以将主要精力放到业务本身。书中使用IPython Notebook作为开发环境，它将代码执行、富文本、公式编辑、绘图、多媒体等功能集合在一起，是科学计算和数据分析的好工具。书中所涉及的数据挖掘对象很丰富，有Iris鸢尾花卉数据集、Ionosphere电离层数据集、NBA比赛结果、MovieLens电影评分数据集、古登堡计划所收集的图书、安然公司邮件数据集、博客语料、CIFAR-10图像数据集等。从这些分属于不同行业的数据集，也能一窥数据挖掘应用之广。此外，作者还介绍了从Twitter、Reddit网站采集数据的方法。在算法方面，除了常见的决策树、朴素贝叶斯、支持向量机等，作者还介绍了最近几年非常热的深度学习。大数据、深度学习对计算能力要求很高，作者介绍了如何在亚马逊云主机上运行MapReduce任务。这本书由浅入深，以真实数据为研究对象，逐渐增大数据集规模，真刀实枪地向读者介绍了Python数据挖掘是怎么回事，并给读者进一步学习指出了多种可能的方向。工程实践之余，作者还不忘介绍数据挖掘常用思路，毫不保留地把自己积攒的宝贵经验传授给读者。这一点我在阅读过程中，深有体会。正如作者自己在前言里所写的，书中不会涉及大量公式推导，

所有的算法都是以很直观的形式向读者介绍，所以即使你缺乏一定的数学基础，只要肯用功，也不用担心自己读不懂。

回到七八年前，当我还是一名英语专业学生的时候，我压根不会想到有一天会学编程，会去翻译这样一本书。后来有幸读了计算机辅助翻译这样一个专业，才开始接触到计算机知识，但是当父亲跟我提起数据建模时，我还是一脸茫然。研究生几年，系里为我们这些非计算机背景的学生开设了Python编程课。从那时起我就有事没事学点Python，一开始是照着*Natural Language Processing with Python*的示例敲，自那时起五年之后我竟想起给NLTK提交几处微小的改动。大约是为了激励我这个后生继续为他们服务，“居心叵测”的Steven Bird竟把我加入到贡献者名单里。去中关村图书大厦的时候，我常常喜欢浏览一下语言与程序设计书架上有没有关于Python的新书，碰到喜欢的就翻翻看，这几年眼看着Python书多了起来，很是欣慰。此外，我去北大、北外旁听过计算语言学、概率统计等课程，去北航旁听过计算机系统基础，看过Udacity的统计学入门和吴恩达老师的机器学习课程视频，兴致来了也曾捧着Rosen的《离散数学及其应用》读上几页。工作中，经常帮同事写个简单的Python程序处理数据，最近还帮他们爬取了一个网站。PyCon北京，我连着去了三四届了，每次都有或多或少的收获，2015年我见过一位大神行云流水般演示用pandas处理数据，很受震撼。以上就是我与Python、数据挖掘的交集。我想说的是，不要再用上学时读的那个专业的思维局限自己的发展，学科的界限在模糊，融合的趋势在增强，数学的重要性在提升。提到数学，今天还看了一个TED演讲视频，说的是借助计算机改变传统数学教育方法。这种理念什么时候能应用到一线教学，非常值得期待。生在这个充满变革的时代，倍感幸运。

以我有限的水平去翻译这样一本书，心里不免焦虑。遇到问题，我四处寻找能人相助。感谢作者Robert Layton，我每次发邮件向他求教或确认问题，他总能很快地回复我，有时第二天回复，还会说抱歉回复晚了。感谢我的同学黄子轩、孙伟、周星，他们在我学习计算机科学的路上给予了很多指导和帮助。翻译本书时，我还向子轩求证作者在第6章给出的示例是否有误。感谢上海大学研究生钱亦欣同学，他帮我审读了第3章，并给出若干很专业的修改意见，上述第6章那处问题，我也曾向他请教过，最终证明是原书弄错了。感谢李少华，他帮我弄明白了Python在Windows系统中的环境变量设置方法。感谢陈健锁，我曾就数据库相关术语向他请教。感谢图灵的朱巍编辑，是她促成了我最终去翻译这样一本趣味盎然的书！感谢图灵公司，我是你们忠实的读者！最后感谢我的妻子，她承担了照顾女儿的重任。女儿的出生，让我惊叹于生命的奇妙！感谢岳父岳母一家人帮忙照看孩子，

我才有时间去做翻译。感谢我的父亲、姐姐，他们以我翻译本书为骄傲。

由于本人学识有限，且时间仓促，书中翻译错误、不当和疏漏之处在所难免，望读者批评指正。

杜春晓

2016年1月3日

前言

你是不是向往数据挖掘的殿堂，却不得其门而入？如果是的话，这本书就是为你而写的。

很多讲授数据挖掘的书涵盖了大量数学知识，倘若读者有较好的数学背景，这自然不错，但我觉得这些书往往只见树木不见森林；也就是说，它们过于关注算法的工作原理，而忘记了我们使用这些算法的初衷。

本书的目标读者是具备一定编程能力、渴望学习数据挖掘的人。我的目标是，如果你认真学完本书，能较好地理解数据挖掘的基础知识，掌握用数据挖掘知识解决问题的最佳实践，此外还能从书中找到几个值得你深入研究的方向。

本书的每一章都会介绍一个新的主题，我会给出该主题的相关算法和数据集。因此，各章主题之间跳跃有点大，阅读本书的过程中需要你的大脑能快速切换。每学完一章，你都要思考一下有没有什么办法能够提升该章中算法的效果，然后尝试去实现它！

我最喜欢的格言中有一条出自莎士比亚的《亨利四世》1：

> 1《亨利四世》是莎士比亚所著的历史剧，分为上下两篇。接下来这句台词出自上篇第三幕第一场，原文为“But will they come when you do call for them?”。遵照朱生豪先生的译法，将“Hotspur”译为“霍茨波”。——译者注

但是你叫他们的时候，他们真的会来吗？

在剧中，有人声称自己会招魂，于是霍茨波（Hotspur）回复了上面这句话。霍茨波的意思是每个人都会招魂，但问题是招过之后，魂来不来。

跟招魂类似，学习数据挖掘无外乎做实验，获得结果。实现一个新的数据挖掘算法或是改善现有实验结果的想法，每个人都能提出来，但真正重要的是你能不能实现它，还有结果是否尽如人意。

本书主要内容

第1章　开始数据挖掘之旅，介绍我们即将用到的技术，接着通过讲

解两个基础算法的实现方法达到热身目的。

第2章　用`scikit-learn`估计器分类，涵盖了数据挖掘的一个重要主题——分类。这一章还会介绍将数据挖掘流程标准化的流水线结构，便于你管理实验流程。

第3章　用决策树预测获胜球队，介绍决策树和随机森林两个新算法。我们将通过抽取区分度高的特征来预测获胜选手。

第4章　用亲和性分析方法推荐电影，思考根据以往消费记录推荐产品的问题，介绍Apriori算法。

第5章　用转换器抽取特征，介绍不同类别特征的抽取方法及不同数据集的处理方法。

第6章　使用朴素贝叶斯进行社会媒体挖掘，使用朴素贝叶斯算法自动分析来自社交网站Twitter的文本信息。

第7章　用图挖掘找到感兴趣的人，采用聚类和网络分析方法，发现社会媒体上感兴趣的人。

第8章　用神经网络破解验证码，从图像中抽取信息，然后训练神经网络，用来发现图像中的单词和字母。

第9章　作者归属问题，通过抽取文本特征，使用支持向量机算法，找出文档的作者。

第10章　新闻语料分类，使用k-means聚类算法，根据新闻文章内容进行分类。

第11章　用深度学习方法为图像中的物体进行分类，采用深度神经网络算法确定图像中的物体。

第12章　大数据处理，探讨对大数据进行数据挖掘的流程及方法。

附录　依次介绍各章的参考资料，便于读者深入了解各章内容。

本书的阅读前提

毫无疑问，要完成本书的学习，你需要准备一台电脑。电脑不能太旧，但也不用配置太高。只要是最近几年的处理器（2010年以后），

4GB内存（RAM）就足够了。在性能稍差一点的机器上运行本书中的大部分代码，应该也不会有问题。

最后两章的代码对机器性能要求较高。在这两章中，我介绍了如何使用亚马逊网络服务（AWS）运行代码。这多少可能会让你破费一些，与在本地运行代码相比，好处是不用做那么多系统配置。如果你不想掏钱购买亚马逊的网络服务，可以在本地主机上配置好所有的工具，当然你需要一台配置较高的计算机：2012年及以后的处理器，内存在4GB以上。

我推荐使用Ubuntu操作系统，但是本书示例代码也可以在Windows、Mac和其他任何Linux系统上运行。然而，你可能需要参考系统的相关文档来完成必要工具的安装。

本书中，我将使用命令行工具`pip`来安装我们用到的Python库。你也可以使用Anaconda，下载地址为http://continuum.io/downloads。

本书所有代码都已在Python 3中测试过。大部分代码无需修改就能在Python 2下运行。如果你遇到什么问题解决不了，请发邮件给我们，我们帮你看看怎么解决。

本书的目标读者

本书是写给那些想把数据挖掘技术应用到实际项目中，却不知道怎么开始的程序员。

如果你没有编程经历，我强烈建议你在学习本书之前，至少先了解一下编程的基础知识。本书直接略过这部分内容，也不会把过多精力放在代码的具体实现上。也就是说，简要学习一下编程基础知识再来学习本书就行。本书不对你的编程技能做过高要求，所以你没必要先成为编程高手！

我强烈建议在阅读本书前最好先积累一些Python编程经验。如果没有的话，也没关系，但是你可能需要先瞧瞧Python代码是怎么回事，可能的话先看看IPython Notebook的教程。用IPython Notebook写程序，跟用其他编辑器写程序（比如使用功能全面的IDE编写Java程序）有一定区别。

排版约定

本书使用不同的文本样式来区分不同类别的内容。以下是常用样式及其用途说明。

首先，我们来看一下最重要的代码所使用的样式。

```
if True:
    print("Welcome to the book")
```

请留意缩进。缩进在Python中表示层级的概念。本书中，我们使用四个空格来表示缩进。你可以按照自己的喜好，使用不同数量的空格（或制表符），但是应该保持一致。如果你被缩进搞糊涂了，请参考本书的示例代码。

在正文中引用代码时，会使用下面这种样式：`I'll use this format`。你不需要在IPython Notebook中输入这些代码，除非我在正文中明确要求你这么做。

所有的命令行输入或输出都使用下面这种样式：

```
# cp file1.txt file2.txt
```

新的术语和重要的词语使用楷体。

此图标表示警告或重要信息。

此图标表示提示或技巧。

读者反馈

我们热忱地欢迎读者朋友给予我们反馈，告诉我们你对于这本书的所思所想——你喜欢或是不喜欢哪些内容。大家的反馈对我们来说至关重要，将能帮助我们确定到底哪些内容是读者真正需要的。

如果你有一般性建议的话，请发邮件至feedback@packtpub.com，请在邮件主题中写清书的名称。

如果你是某一方面的专家，对某个主题特别感兴趣，有意向自己或是与别人合作写一本书，请到www.packtpub.com/authors查阅我们为

作者准备的帮助文档。

客户支持

为自己拥有一本Packt出版的书而自豪吧！为了让你的书物有所值，我们还为你准备了以下内容。

下载示例代码

如果你是从http://www.packtpub.com网站购买的图书，用自己的账号登录后，可以下载所有已购图书的示例代码。如果你是从其他地方购买的，请访问http://www.packtpub.com/support网站并注册，我们会用邮件把代码文件直接发给你。

下载配套PDF文件

我们还为你准备了一个PDF文件，该文件包含书中的所有屏幕截图/图表。这些彩色图像能弥补图书中黑白图像的不足，有助于你理解本书内容。该文件的下载地址为https://www.packtpub.com/sites/default/files/downloads/6053OS_ColorImages.pdf。

勘误表

即使我们竭尽所能来保证图书内容的正确性，错误也在所难免。如果你在我们出版的任何一本书中发现错误——可能是在文本或代码中——倘若你能告诉我们，我们将会非常感激。你的善举足以减少其他读者在阅读出错位置时的纠结和不快，帮助我们在后续版本中更正错误。如果你发现任何错误，请访问http://www.packtpub.com/submit-errata，选择相应书籍，点击“Errata Submission Form”链接，输入错误之处的具体信息。你提交的错误得到验证之后，我们就会接受你的建议，该处错误信息将会上传到我们的网站或是添加到已有勘误表的相应位置。

访问https://www.packtpub.com/books/content/support，在搜索框中输入书名，可查看该图书已有的勘误信息。这部分信息会在Errata部分显示。

侵权

所有媒体在互联网上都面临的一个问题就是侵权。对Packt来说，我们严格保护我们的版权和许可证。如果你在网上发现针对我们出版物的任何形式的盗版产品，请立即告知我们地址或网站名称，以便我们进行补救。

请将盗版书籍的网站地址发送到copyright@packtpub.com。

如果你能这么做，就是在保护我们的作者，保护我们，只有这样我们才能继续以优质内容回馈像你这样热心的读者。

问题

你对本书有任何方面的问题，都可以通过questions@packtpub.com邮箱联系我们，我们也将尽最大努力来帮你答疑解惑。

第 1 章　开始数据挖掘之旅

我们今天的数据采集规模在人类历史上是空前的，日常生活也越来越依赖我们所采集的这些信息。我们希望计算机能把网页翻译成其他语言，预报天气，推荐我们喜欢的书，诊断我们的健康问题。类似的需求还会继续增长，我们会需要更多的应用和更高的精确性。数据挖掘技术可以用来训练计算机，使其根据已有数据做出决策。如今，数据挖掘技术已成为支撑很多高科技系统的骨干。

Python的迅速普及并非偶然。它的灵活度高；模块众多，可以执行很多任务；比起其他任何编程语言，Python代码通常更为简洁，可读性更强。Python在数据挖掘领域已经形成了一个由研究员、从业者和新手组成的氛围活跃的大社区。

本章将介绍如何使用Python进行数据挖掘，主要会涉及以下几个主题。

- 数据挖掘简介及其应用场景
- 搭建Python数据挖掘环境
- 亲和性分析示例：根据购买习惯推荐商品
- （经典）分类问题示例：根据测量结果推测植物的种类

1.1　数据挖掘简介

数据挖掘旨在让计算机根据已有数据做出决策。决策可以是预测明天的天气、拦截垃圾邮件、检测网站的语言，或者在约会网站上发现新的恋爱对象。数据挖掘方面的应用已经有很多，新的应用也在源源不断地出现。

数据挖掘涉及算法、统计学、工程学、最优化理论和计算机科学相关领域的知识。除此之外，我们还会用到语言学、神经科学、城市规划等其他领域的概念或知识。要想充分发挥数据挖掘的威力，通常需要在算法中整合这些属于特定领域的知识。

虽然数据挖掘相关应用的实现细节可能千差万别，但是从较高的层次

看，它们往往大同小异。数据挖掘的第一步一般是创建数据集，数据集能够描述真实世界的某一方面。数据集主要包括以下两个部分。

- 表示真实世界中物体的样本。样本可以是一本书，一张照片，一个动物，一个人或是其他任何物体。
- 描述数据集中样本的特征。特征可以是长度、单词频率、腿的数量、创建时间等。

接下来是调整算法。每种数据挖掘算法都有参数，它们或者是算法自身包含的，或者是使用者添加的。这些参数会影响算法的具体决策。

举个简单的例子，我们希望计算机能够把人按照个子高矮分成两大类。我们首先采集数据，得到包含每个人身高的一组数据，以及对他们高矮的判断。

身高	高矮
155cm	矮
165cm	矮
175cm	高
185cm	高

接下来要做的就是调整我们的算法。作为一个简单的算法，如果身高高于x，我们就认为这个人是高个子，否则，他就属于矮个子。我们的算法要过一遍数据，确定x的最佳值。对于上面的数据集，x 比较合理的值为170cm。任何高于170cm的人就被归到高个子一类中，其余则为矮个子。

在上面这个数据集中，特征显而易见为身高。因为我们想知道人们的高矮，所以采集了他们的身高数据。抽取特征是数据挖掘过程的一个重要环节。本书后面的章节中会介绍从数据集中抽取区分度高的特征的方法。特征抽取往往需要对相关领域有着深入的理解，或至少需要多次试错。

> 本书中使用Python语言介绍数据挖掘。出于讲解的需要，为了保证代码、流程的清晰易懂，我们有时候跳过了能够提升算法速度、效果的细节，没有采用最优方案。

1.2 使用Python和IPython Notebook

本节将介绍Python的安装方法，及本书所用到的开发环境IPython Notebook的搭建方法。此外，还将安装第一部分示例代码所用到的numpy库。

1.2.1 安装Python

Python是一门出色的、应用范围广泛且简单易用的编程语言。

本书将使用Python 3.4版本，你可以根据自己的系统从Python官网https://www.python.org/downloads/下载合适的版本。

Python主要有两大版本Python 3.4和Python 2.7。记得要下载安装Python 3.4，本书所有代码都在该版本中测试过。

本书假定读者了解编程和Python相关知识。本书不要求你是Python编程高手，当然有较多的知识储备学起来更容易。

如果你没有任何编程经验，我建议你先看看《Python学习手册》。

Python官网为新手准备了两份在线教程。

- 非程序员背景，想通过Python学习编程：

  https://wiki.python.org/moin/BeginnersGuide/NonProgrammers

- 程序员背景，想专门学习Python：

  https://wiki.python.org/moin/BeginnersGuide/Programmers

> Windows用户设置好环境变量后，才能在命令行中使用Python。方法如下。首先，找到Python 3的安装路径，默认为C:\Python34。接下来，在命令行（cmd程序）中输入以下命令：将环境设置为`PYTHONPATH=%PYTHONPATH%; C:\Python34`[1]。如果你将Python安装到其他路径下，请根据实际情况调整上述命令中的C:\Python34。

1Python官网介绍了Windows系统的两种环境变量设置方法。建议直接把Python的安装路径添加到Path中，位置如下：计算机—属性—高级系统设置—环境变量，这也是官网介绍的第一种方法。译者使用的就是这一种。作者讲的是第二种方法，详见https://docs.python.org/3.4/using/windows.html#excursus-setting-environment-variables。——译者注

安装好Python，打开命令提示符，输入以下命令：

```
$ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on Linux
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, world!")
Hello, world!
>>> exit()
```

请注意，我们用美元符号（$）表示紧跟在后面的语句是一条命令，要输入到终端（Unix系统中的shell，Windows系统中的cmd程序）。美元符号及后面的空格无须输入。输入后面的内容，然后敲回车执行命令。

运行完经典的“Hello, world!”例子后，退出Python，继续安装用来运行Python代码的更为高级的开发环境IPython Notebook。

> Python 3.4内置了Python的包管理器pip，用它来安装Python包很方便。使用 `$ pip3 freeze`**2** 命令可以验证pip是否能正常运行，该命令还会输出你用它安装过哪些包。

2Windows用户需要事先把pip添加到环境变量中，才能在命令行使用。——译者注

1.2.2　安装IPython

Python开发平台IPython提供多种Python开发工具和开发环境，比标准解释器多出好多功能。IPython Notebook功能强大，有了它，你就可以在Web浏览器中编写程序。它会为代码添加样式，显示运行结果，允许你为代码添加注释。用它来做数据分析再好不过，我们将把它作为主要的开发环境。

请在命令提示符后（注意不是Python中），输入以下命令安装IPython：

```
$ pip install ipython[all]
```

如果要为系统所有用户安装IPython，需要管理员权限。如果你只想自己用或者没有权限做系统级别的变更，则使用以下命令为当前用户

安装即可：

```
$ pip install --user ipython[all]
```

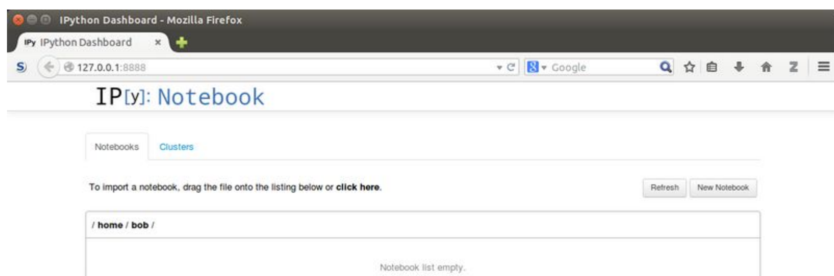
以上命令只为当前用户安装IPython——该系统的其他用户将无法使用。安装过程中若遇到问题，请查阅官方文档，了解更多帮助信息：http://ipython.org/install.html。

安装好IPython Notebook后，运行方式如下：

```
$ ipython3 notebook
```

上述命令帮你做了两件事。首先，在命令提示符界面创建一个IPython Notebook实例。其次，打开Web浏览器，连接到实例，你可以在此创建新的笔记本文件3。Notebook界面如下图所示（注意图中的home/bob为当前用户的主目录，你看到的是自己的主目录，所以目录名称很可能不同）。

3笔记本文件，英文为“notebook”，即用IPython Notebook创建的文件。——译者注



IPython Notebook的关闭方法如下：打开运行实例的终端界面（就是你之前用IPython命令创建Notebook实例的界面），按下Ctrl+C键，系统提示`Shutdown this notebook server (y/[n])?`，询问你是否关闭笔记本服务器。输入`y`，敲回车，`IPython Notebook`就会关闭。

1.2.3　安装**scikit-learn**库

scikit-learn是用Python开发的机器学习库，它包含大量机器学习算法、数据集、工具和框架。它以Python科学计算的相关工具集为基础，其中numpy和scipy等都针对数据处理任务进行过优化，因此scikit-learn速度快、扩展性强，新手会觉得它很好用，老手也不会觉得它功能逊色。更多内容请见第2章。

scikit-learn库可用Python 3提供的pip工具进行安装，之前没有安装NumPy和SciPy的话，也会顺便安装。用管理员/根用户权限打开一个终端，然后输入以下命令：

```
$ pip3 install -U scikit-learn
```

> Windows用户在安装scikitlearn之前，可能需要先安装NumPy和SciPy。安装指南请见www.scipy.org/install.html。

Ubuntu或红帽（Red Hat）等Linux系统的用户也许希望用自带的包管理器安装scikit-learn，但是它们提供的版本很可能不是最新的，所以在安装前需仔细核对版本。本书使用的版本不能低于0.14，否则书中代码可能无法运行。

如何通过编译源文件进行安装，以及更多的安装指南，请见官方文档：

http://scikit-learn.org/stable/install.html。

1.3　亲和性分析示例

终于迎来了第一个数据挖掘的例子，我们拿这个亲和性分析的示例来具体看下数据挖掘到底是怎么回事。数据挖掘有个常见的应用场景，即顾客在购买一件商品时，商家可以趁机了解他们还想买什么，以便把多数顾客愿意同时购买的商品放到一起销售以提升销售额。当商家收集到足够多的数据时，就可以对其进行亲和性分析，以确定哪些商品适合放在一起出售。

1.3.1　什么是亲和性分析

亲和性分析根据样本个体（物体）之间的相似度，确定它们关系的亲疏。亲和性分析的应用场景如下。

- 向网站用户提供多样化的服务或投放定向广告。

- 为了向用户推荐电影或商品，而卖给他们一些与之相关的小玩意。

- 根据基因寻找有亲缘关系的人。

亲和性有多种测量方法。例如，统计两件商品一起出售的频率，或者统计顾客购买了商品1后再买商品2的比率。当然还有别的方法，比如后面章节要讲的计算个体之间的相似度。

1.3.2　商品推荐

商品销售从线下搬到线上后，很多之前靠人工完成的工作只有实现自动化，才有望将生意做大。以向上销售为例，向上销售出自英文up-selling，指的是向已经购买商品的顾客推销另一种商品。原来线下由人工来完成的商品推荐工作，现在依靠数据挖掘技术就能完成，而且每年能为商家多进账几亿美元，强力助推电子商务革命的发展！

我们一起看下简单的商品推荐服务，它背后的思路其实很好理解：人们之前经常同时购买的两件商品，以后也很可能会同时购买。该想法确实很简单吧，可这就是很多商品推荐服务的基础，无论线上还是线下。

这种想法很容易转化为算法。顾客购买商品后，在向他们推荐商品前，先查询一下历史交易数据，找到以往他们购买同样商品的交易数据，看看同时购买了什么，再把它们推荐给顾客即可。该算法实际表现也不错，至少比随机推荐商品更有效。然而，它还有很大的提升空间，这正是数据挖掘一展身手的好机会。

为了简化代码，方便讲解，我们只考虑一次购买两种商品的情况。例如，人们去超市既买了面包，又买了牛奶。作为数据挖掘入门性质的例子，我们希望得到下面这样的规则：

> 如果一个人买了商品X，那么他很有可能购买商品Y。

多件商品的规则会更为复杂，比如购买香肠和汉堡包的顾客比起其他顾客更有可能购买番茄酱，本书中不涉及这样的规则。

1.3.3　在NumPy中加载数据集

下载本书配套代码包，保存到你的计算机上，然后找到这个例子的数据集。本例中，建议你新建一个文件夹，把数据集和代码都放进去。在当前目录4下，启动IPython Notebook，导航进入新建的文件夹，创建一个新的笔记本文件。

> 4在命令行，切换到新建的文件夹，输入ipython3 notebook命令。——译者注

处理该数据集要用到NumPy的二维数组，书中大部分例子都会用到这种数据结构。数组看上去像是一张表，每一行表示样本中一个个体，每一列表示一种特征。

数组的每一项为个体的某项特征值。说起来有些拗口，为方便讲解，使用如下代码把数据集加载进来，稍后输出数组的部分数据看看效果：

```
import numpy as np
dataset_filename = "affinity_dataset.txt"
X = np.loadtxt(dataset_filename)
```

运行IPython Notebook，创建笔记本文件，在第一个格子中输入上述代码。按下Shift+Enter（同时创建新的格子）运行代码。代码运行完毕后，第一个格子左侧的方括号中出现一个表示序号的数字，看到这个数字就表明程序运行结束。第一个格子应该如下所示：

```
In [1]: import numpy as np
        dataset_filename = "affinity_dataset.txt"
        X = np.loadtxt(dataset_filename)
```

对于笔记本文件，前面的代码运行完后，后面的才能运行；还没有轮到它运行或是在运行中时，方括号中显示一个星号。运行结束后，星号立刻变为序号。

记得把数据集文件和笔记本文件放到同一目录下。否则，请修改上述代码中dataset_filename变量的值。

接下来，我们看看数据集到底是什么样子。在笔记本空格子中输入以下代码，输出数据集的前5行看看：

```
print(X[:5])
```

如果你从http://www.packtpub.com网站购买的图书，登录后即可下载已购图书的代码文件。如果你是从别处购买的图书，访问http://www.packtpub.com/support，注册后，我们可以用电子邮件把你需要的文件发给你。5

5注册后，可自行下载。——译者注

上述代码的运行结果为前5次交易中顾客都买了什么。

```
In [2]: print(X[:5])

[[ 0.  0.  1.  1.  1.]
 [ 1.  1.  0.  1.  0.]
 [ 1.  0.  1.  1.  0.]
 [ 0.  0.  1.  1.  1.]
 [ 0.  1.  0.  0.  1.]]
```

输出结果从横向和纵向看都可以。横着看，每次只看一行。第一行（0, 0, 1, 1, 1）表示第一条交易数据所包含的商品。竖着看，每一列代表一种商品。在我们这个例子中，这五种商品分别是面包、牛奶、奶酪、苹果和香蕉。从第一条交易数据中，我们可以看到顾客购买了奶酪、苹果和香蕉，但是没有买面包和牛奶。

每个特征只有两个可能的值，1或0，表示是否购买了某种商品，而不是购买商品的数量。1表示顾客至少买了1个单位的该商品，0表示顾客没有买该种商品。

1.3.4 实现简单的排序规则

正如之前所说，我们要找出“如果顾客购买了商品X，那么他们可能愿意购买商品Y”这样的规则6。简单粗暴的做法是，找出数据集中所有同时购买的两件商品。找出规则后，还需要判断其优劣，我们挑好的规则用。

6一条规则由前提条件和结论两部分组成。——译者注

规则的优劣有多种衡量方法，常用的是支持度（support）和置信度（confidence）。

支持度指数据集中规则应验的次数，统计起来很简单。有时候，还需要对支持度进行规范化，即再除以规则有效前提下的总数量。我们这里只是简单统计规则应验的次数。

支持度衡量的是给定规则应验的比例，而置信度衡量的则是规则准确率如何，即符合给定条件（即规则的“如果”语句所表示的前提条件）的所有规则里，跟当前规则结论一致的比例有多大。计算方法为首先统计当前规则的出现次数，再用它来除以条件（“如果”语句）相同的规则数量。

接下来，通过一个例子来说明支持度和置信度的计算方法，我们看一下怎么求“如果顾客购买了苹果，他们也会购买香蕉”这条规则的支持度和置信度。

如下面的代码所示，通过判断交易数据中`sample[3]`的值，就能知道一个顾客是否买了苹果。这里，sample表示一条交易信息，也就是数据集里的一行数据。

```
In [9]: # First, how many rows contain our premise: that a person is buying apples
        num_apple_purchases = 0
        for sample in X:
            if sample[3] == 1:  # This person bought Apples
                num_apple_purchases += 1
        print("{0} people bought Apples".format(num_apple_purchases))

        36 people bought Apples
```

同理，检测`sample[4]`的值是否为1，就能确定顾客有没有买香蕉。现在可以计算题目给定规则在数据集中的出现次数，从而计算置信度和支持度。

我们需要统计数据集中所有规则的相关数据。首先分别为规则应验和规则无效这两种情况创建字典。字典的键是由条件和结论组成的元组，元组元素为特征在特征列表中的索引值，不要用实际特征名，比如“如果顾客购买了苹果，他们也会买香蕉”就用(3, 4)表示。如果某个个体的条件和结论均与给定规则相符，就表示给定规则对该个体适用，否则如果通过给定条件推出的结论与给定规则的结论不符，则表示给定规则对该个体无效。

为了计算所有规则的置信度和支持度，首先创建几个字典，用来存放计算结果。这里使用`defaultdict`，好处是如果查找的键不存在，返回一个默认值。需要统计的量有规则应验、规则无效及条件相同的规则数量。

```
from collections import defaultdict
```

```
valid_rules = defaultdict(int)
invalid_rules = defaultdict(int)
num_occurances = defaultdict(int)
```

计算过程需要用到循环结构，依次对样本的每个个体及个体的每个特征值进行处理。第一个特征为规则的前提条件——顾客购买了某一种商品。

```
for sample in X:
  for premise in range(5):
```

检测个体是否满足条件，如果不满足，继续检测下一个条件。

```
    if sample[premise] == 0: continue
```

如果条件满足（即值为1），该条件的出现次数加1。在遍历过程中跳过条件和结论相同的情况，比如“如果顾客买了苹果，他们也买苹果”，这样的规则没有多大用处7。

> 7n_samples, n_features = X.shape，详见本书配套代码。——译者注

```
    num_occurances[premise] += 1
    for conclusion in range(n_features):
        if premise == conclusion: continue
```

如果规则适用于个体，规则应验这种情况（valid_rules字典中，键为由条件和结论组成的元组）增加一次，反之，违反规则情况（invalid_rules字典中）就增加一次。

```
    if sample[conclusion] == 1:
      valid_rules[(premise, conclusion)] += 1
    else:
      invalid_rules[(premise, conclusion)] += 1
```

得到所有必要的统计量后，我们再来计算每条规则的支持度和置信度。如前所述，支持度就是规则应验的次数。

```
support = valid_rules
```

置信度的计算方法类似，遍历每条规则进行计算。

```
confidence = defaultdict(float)
for premise, conclusion in valid_rules.keys():
    rule = (premise, conclusion)
    confidence[rule] = valid_rules[rule] / num_occurances[premise]
```

我们得到了支持度字典和置信度字典，分别包含每条规则的支持度和置信度。我们再来声明一个函数，接收的参数有：分别作为前提条件和结论的特征索引值、支持度字典、置信度字典以及特征列表。输出每条规则及其支持度和置信度，对输出进行格式化，以方便查看。

```
def print_rule(premise, conclusion,
              support, confidence, features):
```

之前建立的features列表派上用场了，每条规则的条件、结论就是用features列表中特征的索引来表示的。输出时，把索引替换成相应的特征，更容易读懂。

```
    premise_name = features[premise]
    conclusion_name = features[conclusion]
    print("Rule: If a person buys {0} they will also buy
      {1}".format(premise_name, conclusion_name))
```

接着输出规则的支持度和置信度。

```
    print(" - Support: {0}".format(support[(premise,
                                            conclusion)]))
    print(" - Confidence: {0:.3f}".format(confidence[(premise,
                                                       conclusion)]))
```

写完后，自己测试一下代码是否可用——尝试更换条件和结论，看看输出结果如何。

```
In [31]: premise = 1
         conclusion = 3
         print_rule(premise, conclusion, support, confidence, features)

         Rule: If a person buys milk they will also buy apples
          - Confidence: 0.196
          - Support: 9
```

1.3.5 排序找出最佳规则

得到所有规则的支持度和置信度后，为了找出最佳规则，还需要根据支持度和置信度对规则进行排序，我们分别看一下这两个标准。

要找出支持度最高的规则，首先对支持度字典进行排序。字典中的元素（一个键值对）默认为没有前后顺序；字典的`items()`函数返回包含字典所有元素的列表。我们使用`itemgetter()`类作为键，这样就可以对嵌套列表进行排序。`itemgetter(1)`表示以字典各元素的值（这里为支持度）作为排序依据，`reverse=True`表示降序排列。

```
from operator import itemgetter
sorted_support = sorted(support.items(), key=itemgetter(1), re
verse=True)
```

排序完成后，就可以输出支持度最高的前5条规则。

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_support[index][0]
    print_rule(premise, conclusion, support, confidence, features)
```

结果如下所示：

```
In [40]: for index in range(5):
             print("Rule #{0}".format(index + 1))
             (premise, conclusion) = sorted_support[index][0]
             print_rule(premise, conclusion, support, confidence, features)
```

```
Rule #1
Rule: If a person buys cheese they will also buy bananas
 - Confidence: 0.659
 - Support: 27

Rule #2
Rule: If a person buys bananas they will also buy cheese
 - Confidence: 0.458
 - Support: 27

Rule #3
Rule: If a person buys apples they will also buy cheese
 - Confidence: 0.694
 - Support: 25

Rule #4
Rule: If a person buys cheese they will also buy apples
 - Confidence: 0.610
 - Support: 25

Rule #5
Rule: If a person buys bananas they will also buy apples
 - Confidence: 0.356
 - Support: 21
```

同理，我们还可以输出置信度最高的规则。首先根据置信度进行排序。

```
sorted_confidence = sorted(confidence.items(), key=itemgetter(1),
reverse=True)
```

再次输出看看结果。注意输出方法相同，但是请留意下面第三行代码里sorted_confidence的变化，不要继续使用sorted_support。

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    premise, conclusion = sorted_confidence[index][0]
    print_rule(premise, conclusion, support, confidence, features)
```

```
In [42]: for index in range(5):
             print("Rule #{0}".format(index + 1))
             (premise, conclusion) = sorted_confidence[index][0]
             print_rule(premise, conclusion, support, confidence, features)
```

```
Rule #1
Rule: If a person buys apples they will also buy cheese
 - Confidence: 0.694
 - Support: 25

Rule #2
Rule: If a person buys cheese they will also buy bananas
 - Confidence: 0.659
 - Support: 27

Rule #3
Rule: If a person buys bread they will also buy bananas
 - Confidence: 0.630
 - Support: 17

Rule #4
Rule: If a person buys cheese they will also buy apples
 - Confidence: 0.610
 - Support: 25

Rule #5
Rule: If a person buys apples they will also buy bananas
 - Confidence: 0.583
 - Support: 21
```

从排序结果来看，“顾客买苹果，也会买奶酪”和“顾客买奶酪，也会买香蕉”，这两条规则的支持度和置信度都很高。超市经理可以根据这些规则来调整商品摆放位置。例如，如果本周苹果促销，就在旁边摆上奶酪。但是香蕉和奶酪同时搞促销就没有多大意义了，因为我们发现购买奶酪的顾客中，接近66%的人即使不搞促销也会买香蕉——即使搞促销，也不会给销量带来多大提升。

从上面这个例子就能看出数据挖掘的洞察力有多强大。人们可以用数据挖掘技术探索数据集中各变量之间的关系，寻找新发现。接下来一节，我们看看数据挖掘的另一个功能：预测。

1.4　分类问题的简单示例

在上述亲和性分析例子中，我们寻找的是数据集中不同变量之间的相关性。而分类问题，只关注类别（也叫作目标）这个变量。上述例子中，假如我们关心的是怎样让顾客买更多的苹果，就可以把是否购买苹果作为类别，使用分类方法，只寻找促成顾客购买苹果的规则。

1.5　什么是分类

分类是数据挖掘领域最为常用的方法之一，不论是实际应用还是科

研，都少不了它的身影。对于分类问题，我们通常能拿到表示实际对象或事件的数据集，我们知道数据集中每一条数据所属的类别，这些类别把一条条数据划分为不同的类。什么是类别？类别的值又是怎么回事？我们来看下面几个例子。

- 根据检测数据确定植物的种类。类别的值为“植物属于哪个种类？”。
- 判断图像中有没有狗。类别是“图像里有狗吗？”。
- 根据化验结果，判断病人有没有患上癌症。类别是“病人得癌症了吗？”。

上述三个问题中有两个是二值（是/否）问题，但正如第一个确定植物类别的问题，多个类别的情况也很常见。

分类应用的目标是，根据已知类别的数据集，经过训练得到一个分类模型，再用模型对类别未知的数据进行分类。例如，我们可以对收到的邮件进行分类，标注哪些是自己希望收到的，哪些是垃圾邮件，然后用这些数据训练分类模型，实现一个垃圾邮件过滤器，这样以后再收到邮件，就不用自己去确认它是不是垃圾邮件了，过滤器就能帮你搞定。

1.5.1　准备数据集

我们接下来将使用著名的Iris植物分类数据集。这个数据集共有150条植物数据，每条数据都给出了四个特征：sepal length、sepal width、petal length、petal width（分别表示萼片和花瓣的长与宽），单位均为cm。这是数据挖掘中的经典数据集之一（1936年就用到了数据挖掘领域！）。该数据集共有三种类别：Iris Setosa（山鸢尾）、Iris Versicolour（变色鸢尾）和Iris Virginica（维吉尼亚鸢尾）。我们这里的分类目的是根据植物的特征推测它的种类。

`scikit-learn`库内置了该数据集，可直接导入。

```
from sklearn.datasets import load_iris
dataset = load_iris()
X = dataset.data
y = dataset.target
```

用`print(dataset.DESCR)`命令查看数据集，大概了解一下，包括

特征的说明。

数据集中各特征值为连续型，也就是有无数个可能的值。测量得到的数据就是这个样子，比如，测量结果可能是1、1.2或1.25，等等。连续值的另一个特点是，如果两个值相近，表示相似度很大。一种萼片长1.2cm的植物跟一种萼片宽1.25cm的植物很相像。

与此相反，类别的取值为离散型。虽然常用数字表示类别，但是类别值不能根据数值大小比较相似性。Iris数据集用不同的数字表示不同的类别，比如类别0、1、2分别表示Iris Setosa、Iris Versicolour、Iris Virginica。但是这不能说明前两种植物，要比第一种和第三种更相近——尽管单看表示类别的数字时确实如此。在这里，数字表示类别，只能用来判断两种植物是否属于同一种类别，而不能说明是否相似。

当然，还有其他类型的特征，后续章节会讲到其中几种。

数据集的特征为连续值，而我们即将使用的算法使用类别型特征值，因此我们需要把连续值转变为类别型，这个过程叫作离散化。

最简单的离散化算法，莫过于确定一个阈值，将低于该阈值的特征值置为0，高于阈值的置为1。我们把某项特征的阈值设定为该特征所有特征值的均值。每个特征的均值计算方法如下。

```
attribute_means = X.mean(axis=0)
```

我们得到了一个长度为4的数组，这正好是特征的数量。数组的第一项是第一个特征的均值，以此类推。接下来，用该方法将数据集打散，把连续的特征值转换为类别型。

```
X_d = np.array(X >= attribute_means, dtype='int')
```

后面的训练和测试，都将使用新得到的X_d数据集（打散后的数组X），而不再使用原来的数据集（X）。

1.5.2　实现OneR算法

OneR算法的思路很简单，它根据已有数据中，具有相同特征值的个体最可能属于哪个类别进行分类。OneR是One Rule（一条规则）的

简写，表示我们只选取四个特征中分类效果最好的一个用作分类依据。后续章节中的分类算法比起OneR要复杂很多，但这个看似不起眼的简单算法，在很多真实数据集上表现得也不凡。

算法首先遍历每个特征的每一个取值，对于每一个特征值，统计它在各个类别中的出现次数，找到它出现次数最多的类别，并统计它在其他类别中的出现次数。

举例来说，假如数据集的某一个特征可以取0或1两个值。数据集共有三个类别。特征值为0的情况下，A类有20个这样的个体，B类有60个，C类也有20个。那么特征值为0的个体最可能属于B类，当然还有40个个体确实是特征值为0，但是它们不属于B类。将特征值为0的个体分到B类的错误率就是40%，因为有40个这样的个体分别属于A类和C类。特征值为1时，计算方法类似，不再赘述；其他各特征值最可能属于的类别及错误率的计算方法也一样。

统计完所有的特征值及其在每个类别的出现次数后，我们再来计算每个特征的错误率。计算方法为把它的各个取值的错误率相加，选取错误率最低的特征作为唯一的分类准则（OneR），用于接下来的分类。

现在，我们就来实现该算法。首先创建一个函数，根据待预测数据的某项特征值预测类别，并给出错误率。在这之前需要导入前面用过的`defaultdict`和`itemgetter`模块。

```
from collections import defaultdict
from operator import itemgetter
```

下面创建函数声明，参数分别是数据集、类别数组、选好的特征索引值、特征值。

```
def train_feature_value(X, y_true, feature_index, value):
```

接下来遍历数据集中每一条数据（代表一个个体），统计具有给定特征值的个体在各个类别中的出现次数。

```
    class_counts = defaultdict(int)
    for sample, y in zip(X, y_true):
        if sample[feature_index] == value:
            class_counts[y] += 1
```

```
```

对class_counts字典进行排序，找到最大值，就能找出具有给定特征值的个体在哪个类别中出现次数最多。

```
    sorted_class_counts = sorted(class_counts.items(),
      key=itemgetter(1), reverse=True)
        most_frequent_class = sorted_class_counts[0][0]
```

接着计算该条规则的错误率。OneR算法会把具有该项特征值的个体统统分到上面找到的出现次数最多的类别中。错误率为具有该特征值的个体在其他类别（除出现次数最多的类别之外的）中的出现次数，它表示的是分类规则不适用的个体的数量。

```
incorrect_predictions = [class_count for class_value, class_count
                         in class_counts.items()
                         if class_value != most_frequent_class]
error = sum(incorrect_predictions)
```

最后返回使用给定特征值得到的待预测个体的类别和错误率。

```
    return most_frequent_class, error
```

对于某项特征，遍历其每一个特征值，使用上述函数，就能得到预测结果和每个特征值所带来的错误率，然后把所有错误率累加起来，就能得到该特征的总错误率。我们来定义一个函数，实现这些操作。

函数声明如下，这次只用到三个参数，上面已经介绍过。

```
def train_on_feature(X, y_true, feature_index):
```

接下来找出给定特征共有几种不同的取值。下面这行代码X[:,feature_index]以数组的形式返回由feature_index所指的列。然后用set函数将数组转化为集合，从而找出有几种不同的取值。

```
    values = set(X[:,feature_index])
```

再创建字典predictors，用作预测器。字典的键为特征值，值为类别。比如键为1.5、值为2，表示特征值为1.5的个体属于类别2。创建

errors列表，存储每个特征值的错误率。

```
    predictors = {}
    errors = []
```

函数的主干部分遍历选定特征的每个不同的特征值，用前面定义的train_feature_value()函数找出每个特征值最可能的类别，计算错误率，并将其分别保存到预测器predictors和errors中。

```
    for current_value in values:
      most_frequent_class, error = train_feature_value(X,
        y_true, feature_index, current_value)
      predictors[current_value] = most_frequent_class
      errors.append(error)
```

最后，计算该规则的总错误率，返回预测器及总错误率。

```
    total_error = sum(errors)
    return predictors, total_error
```

1.5.3 测试算法

上一节中亲和性分析算法的目标是从数据集中发现用以指导实践的规则。而分类问题有所不同，我们想建立一个能够根据已有知识对没有见过的个体进行分类的模型。

我们因此把机器学习流程分为两步：训练和测试。在训练阶段，我们从数据集中取一部分数据，创建模型。在测试阶段，我们测试模型在数据集上的分类效果。考虑到模型的目标是对新个体进行分类，因此不能用测试数据训练模型，因为这样做容易导致过拟合问题。

过拟合指的是模型在训练集上表现很好，但对于没有见过的数据表现很差。解决方法很简单：千万不要用训练数据测试算法。详细的处理方法很复杂，后续章节会有所涉及；我们这里简单化处理，把数据集分为两个小部分，分别用于训练和测试。具体流程接下来会介绍。

scikit-learn库提供了一个将数据集切分为训练集和测试集的函数。

```
from sklearn.cross_validation import train_test_split
```

该函数根据设定的比例（默认把数据集的25%作为测试集）将数据集随机切分为两部分，以确保测试结果的可信度。

```
X_train, X_test, y_train, y_test = train_test_split(X_d, y, random_state=1
```

这样我们就得到了两个数据集：训练集X_train和测试集X_test。y_train和y_test分别为以上两个数据集的类别信息。

切分函数的第三个参数random_state用来指定切分的随机状态。每次切分，使用相同的随机状态，切分结果相同。虽然看起来是随机的，但是它所使用的算法是确定的，输出结果也是一致的。在书中所有用到random_state的地方，我建议你跟我使用同一个值，这样你得到的结果就应该跟我的相同，这样便于你验证结果。把random_state的值设置为none，每次切分结果将是真正随机的。

接下来，计算所有特征值的目标类别（预测器）。记得只使用训练集。遍历数据集中的每个特征，使用我们先前定义的函数train_on_feature()训练预测器，计算错误率。

```
all_predictors = {}
errors = {}
for feature_index in range(X_train.shape[1]):
  predictors, total_error = train_on_feature(X_train, y_train,
    feature_index)
  all_predictors[feature_index] = predictors
  errors[feature_index] = total_error
```

然后找出错误率最低的特征，作为分类的唯一规则。

```
best_variable, best_error = sorted(errors.items(), key=itemgetter(1))
[0]
```

对预测器进行排序，找到最佳特征值，创建model模型。

```
model = {'feature': best_variable,
  'predictor': all_predictors[best_variable]}
```

model模型是一个字典结构，包含两个元素：用于分类的特征和预测

器。有了模型后，就可以根据特征值对没有见过的数据进行分类。示例如下：

```
variable = model['variable']
predictor = model['predictor']
prediction = predictor[int(sample[variable])]
```

我们经常需要一次对多条数据进行预测，为此用上面的代码实现了下面这个函数，通过遍历数据集中的每条数据来完成预测。

```
def predict(X_test, model):
    variable = model['variable']
    predictor = model['predictor']
    y_predicted = np.array([predictor[int(sample[variable])] for
      sample in X_test])
    return y_predicted
```

我们用上面这个函数预测测试集中每条数据的类别。

```
y_predicted = predict(X_test, model)
```

比较预测结果和实际类别，就能得到正确率是多少。

```
accuracy = np.mean(y_predicted == y_test) * 100
print("The test accuracy is {:.1f}%".format(accuracy))
```

输出结果为65.8%，对于只使用一条规则来说，这就很不错了！

1.6　小结

本章介绍了如何用Python进行数据挖掘。如果你能运行这一部分的代码8（见代码包第1章的文件夹），说明开发环境已搭建好，后续章节的大部分代码都能运行了。当然有些Python库还没装，随用随装就好。

> 8如果.ipynb文件在Notebook中打开时报错，请用JSON检查工具查找有无不合法的JSON字符，自行调整一下。——译者注

我们用IPython Notebook运行了代码，好处是能及时看到一小块代码

的输出。它功能强大，后面会继续使用。

我们举了一个简单的亲和性分析的例子，用它找出顾客经常一起购买的商品。这种探索性的分析方法用处很大，能帮助人们发现商业流程、某个环境或场景中的潜在规律。亲和性分析可用在商业、医疗、人工智能等领域，说不定能这些领域带来突破。

本章还通过OneR算法介绍了分类的应用。该算法寻找最佳的特征值用于分类，该特征值在训练集中哪个类别中出现的次数最多，待预测数据就属于哪个类别。

后续章节会扩展分类和亲和性分析的概念，同时还会介绍`scikit-learn`库以及它实现的一些数据挖掘算法。

第 2 章　用scikit-learn估计器分类

用Python语言编写的scikit-learn库，实现了一系列数据挖掘算法，提供通用编程接口、标准化的测试和调参工具，便于用户尝试不同算法对其进行充分测试和查找最优参数值。有大量使用scikit-learn库的算法和工具。

本章讲解数据挖掘通用框架的搭建方法。有了这样一个框架，后续章节就可以把讲解重点放到数据挖掘应用和技术上面。

本章主要介绍如下几个概念。

- 估计器（Estimator）：用于分类、聚类和回归分析。
- 转换器（Transformer）：用于数据预处理和数据转换。
- 流水线（Pipeline）：组合数据挖掘流程，便于再次使用。

2.1　scikit-learn估计器

为帮助用户实现大量分类算法，scikit-learn把相关功能封装成所谓的估计器。估计器用于分类任务，它主要包括以下两个函数。

- `fit()`：训练算法，设置内部参数。该函数接收训练集及其类别两个参数。
- `predict()`：参数为测试集。预测测试集类别，并返回一个包含测试集各条数据类别的数组。

大多数scikit-learn估计器接收和输出的数据格式均为numpy数组或类似格式。

scikit-learn提供了大量估计器，其中有支持向量机（SVM）、随机森林、神经网络等，多数算法本书都会有所涉及。本章先介绍scikit-learn中的近邻算法。

我们需要安装matplotlib库，作图时会用到。最简单的安装方法就是用pip3来安装，在第1章安装scikitlearn时用过。

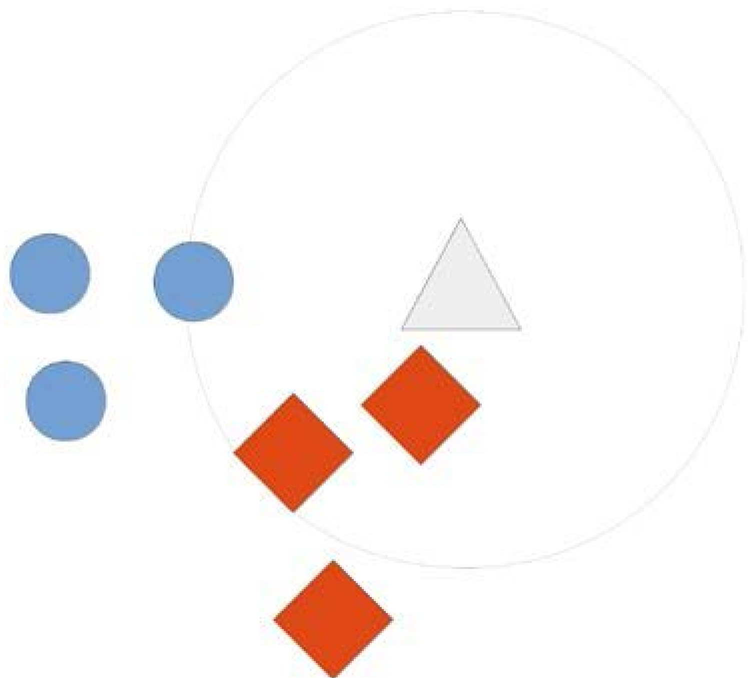
```
$pip3 install matplotlib
```

安装过程中若遇到任何问题，请参考官方给出的安装指南：http://matplotlib.org/users/installing.html。

2.1.1　近邻算法

近邻算法可能是标准数据挖掘算法中最为直观的一种。为了对新个体进行分类，它查找训练集，找到与新个体最相似的那些个体，看看这些个体大多属于哪个类别，就把新个体分到哪个类别。

举例来说，我们要根据三角形更像什么（跟哪种图形离得更近），预测三角形的类别。我们找到三个离它最近的邻居：两个菱形和一个圆。菱形的数量多于圆，因此我们预测三角形的类别为菱形。



近邻算法几乎可以对任何数据集进行分类，但是，要计算数据集中每

两个个体之间的距离，计算量很大。例如，数据集中个体数量为10时，需要计算45对不同个体之间的距离。然而，当个体数量为1000时，要计算大约50万对个体之间的距离！现在有很多提升该算法速度的方法，后续章节会讲到几种。

除了计算量大之外，该算法还有一个问题，就是在特征取离散值的数据集上表现很差。遇到这种情况，应该考虑使用其他算法。

2.1.2　距离度量

距离是数据挖掘的核心概念之一。我们往往需要知道两个个体之间的距离是多少。更进一步说，我们还得能够解决一对个体相对另一对个体是否更相近等问题。这类问题的解决方法，将直接影响分类结果。

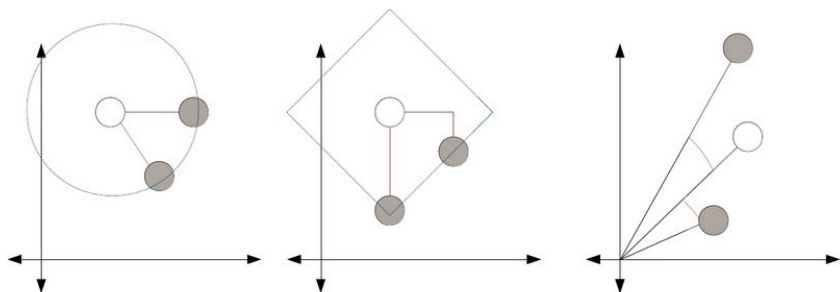
人们耳熟能详的距离度量方法就是欧氏距离，即真实距离。假如你在图像中画两个点，用直尺测量这两个点之间的距离，得到的结果就是欧氏距离。稍微正式点来说，它其实是两个特征向量长度平方和的平方根。

欧氏距离确实很直观，但是如果某些特征比其他特征取值大很多，精确度就会比较差。此外，如果很多特征值为0，也就是所谓的稀疏矩阵，结果也不准确。这时可以用其他距离度量方法，常用的有曼哈顿距离和余弦距离。

曼哈顿距离为两个特征在标准坐标系中绝对轴距之和（没有使用平方距离）。拿国际象棋中的车1举例子，这样更形象。假如车每次只能走一格，那么它走到当前格子对角线那头，所走的距离就是曼哈顿距离。虽然异常值也会影响分类结果，但是其所受的影响要比欧氏距离小得多。

> 1国际象棋中车的走法跟我国象棋中车的走法相同，只能沿水平或垂直方向移动。——译者注

余弦距离更适合解决异常值和数据稀疏问题。直观上讲，余弦距离指的是特征向量夹角的余弦值。下面为以上三种距离的示意图。



在上面每张图中，两个灰圆与白圆之间的距离是相等的。左图是欧氏距离，因此两个灰圆都落在以白圆为圆心的同一个圆的圆周上，可以用尺子量量看。中间这幅图表示的是曼哈顿距离，它指的是从灰圆到白圆所走的横向和纵向距离之和，也叫街区距离（City Block），测量时要想象棋中车的走法。右图是余弦距离示意图，计算夹角之间的余弦值，忽略特征向量的长度。

采用哪种距离度量方法对最终结果有很大影响。例如，你的数据集有很多特征，但是如果任意一对个体之间的欧氏距离都相等，那么你就没法通过欧氏距离进行比较了！曼哈顿距离在某些情况下具有更高的稳定性，但是如果数据集中某些特征值很大，用曼哈顿距离的话，这些特征会掩盖其他特征间的邻近关系。最后，再来说说余弦距离，它适用于特征向量很多的情况，但是它丢弃了向量长度所包含的在某些场景下可能会很有用的一些信息。

本章，我们主要介绍欧氏距离，其他距离后面章节再介绍。

2.1.3　加载数据集

即将用到的数据集叫作电离层（Ionosphere），这些数据是由高频天线收集的。这些天线的目的是侦测在电离层和高层大气中存不存在由自由电子组成的特殊结构。如果一条数据能给出特殊结构存在的证据，这条数据就属于好的那一类（在数据集中用“g”表示），否则就是坏的（用“b”表示）。我们要做的就是建立分类器，自动判断这些数据的好坏。



（图像来自**https://www.flickr.com/photos/geckzilla/16149273389/**）

Ionosphere数据集可以从UCI机器学习数据库下载，该数据库包含大量数据集，可用于多种数据挖掘任务。打开http://archive.ics.uci.edu/ml/datasets/Ionosphere，点击Data Folder。在随后打开的页面中，下载ionosphere.data和ionosphere.names文件。把这两个文件保存到用户主目录下的Data文件夹中。



主目录的位置取决于操作系统。Windows系统中通常为C:\Documents and Settings\username。Mac和Linux系统中为/home/username。如果你不确定，可以使用下面的Python代码输出主目录所在位置：

```
import os
print(os.path.expanduser("~"))
```

该数据集每行有35个值，前34个为17座天线采集的数据（每座天线采集两个数据）。最后一个值不是“g”就是“b”，表示数据的好坏，即是否提供了有价值的信息。

启动IPython Notebook服务器，新建名为Ionosphere Nearest Neighbors的笔记本文件。

首先，导入numpy和csv库，下面会用到。

```
import numpy as np
import csv
```

加载数据集前，用Data文件夹路径、数据集所在的文件夹名称和数据集名称组合成数据集文件的完整路径。

```
data_filename = os.path.join(data_folder, "Ionosphere",
"ionosphere.data")
```

创建Numpy数组X和y存放数据集。数据集大小已知，共有351行34列。在以后的实际工作中，如果你不知道数据集大小也没关系——后面章节会讲如何在不知道数据集大小的情况下加载它，具体怎么做现在不知道也没关系。

```
X = np.zeros((351, 34), dtype='float')
y = np.zeros((351,), dtype='bool')
```

Ionosphere数据集文件为CSV（Comma-Separated Values，用逗号分隔数据项）格式，这是常用的数据集存储格式。我们用csv模块来导入数据集文件，并创建csv阅读器对象。

```
with open(data_filename, 'r') as input_file:
    reader = csv.reader(input_file)
```

接着，遍历文件中的每一行数据。每行数据代表一组测量结果，我们可以将其称作数据集中的一个个体。用枚举函数来获得每行的索引号，在下面更新数据集X中的某一个体时会用到行号。

```
    for i, row in enumerate(reader):
```

获取每一个个体的前34个值，将其强制转化为浮点型，保存到X中。

```
    data = [float(datum) for datum in row[:-1]]
    X[i] = data
```

最后，获取每个个体最后一个表示类别的值，把字母转化为数字，如果类别为“g”，值为1，否则值为0。

```
    y[i] = row[-1] == 'g'
```

到此，我们就把数据集读到了数组x中，类别读入了数组y中，与我们在上一章的做法相同。

2.1.4 努力实现流程标准化

正如本章开头讲过的，scikit-learn估计器由两大函数组成：fit()和predict()。用fit方法在训练集上完成模型的创建，用predict方法在测试集上评估效果。

首先，需要创建训练集和测试集。导入并运行train_test_split函数。

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=14)
```

然后，导入 K 近邻分类器这个类，并为其初始化一个实例。现阶段，参数用默认的即可，后面再讲参数调优。该算法默认选择5个近邻作为分类依据。

```
from sklearn.neighbors import KNeighborsClassifier
estimator = KNeighborsClassifier()
```

估计器创建好后，接下来就要用训练数据进行训练。K 近邻估计器分析训练集中的数据，比较待分类的新数据点和训练集中的数据，找到新数据点的近邻。

```
estimator.fit(X_train, y_train)
```

接着，用测试集测试算法，评估它在测试集上的表现。

```
y_predicted = estimator.predict(X_test)
accuracy = np.mean(y_test == y_predicted) * 100
print("The accuracy is {0:.1f}%".format(accuracy))
```

正确率为86.4%。使用默认参数，只用少数几行代码就能达到这个效果，真是很厉害！虽然scikit-learn提供的大多数默认参数适用范围广，效果也不错，但我们还是要学着根据实验的实际情况，尽可能选用合适的参数值，争取达到最佳效果。

2.1.5 运行算法

在先前的几个实验中，我们把数据集分为训练集和测试集，用训练集训练算法，在测试集上评估效果。倘若碰巧走运，测试集很简单，我们就会觉得算法表现很出色。反之，我们可能会怀疑算法很糟糕。也许由于我们一时不走运，就把一个其实很不错的算法给无情抛弃了，这岂不是很可惜。

交叉检验能解决上述一次性测试所带来的问题。既然只切一次有问题，那就多切几次，多进行几次实验。每次切分时，都要保证这次得到的训练集和测试集与上次不一样，还要确保每条数据都只能用来测试一次。算法描述如下。

(1) 将整个大数据集分为几个部分（fold2）。

> 2行话叫“折”。——译者注

(2) 对于每一部分执行以下操作：

- 将其中一部分作为当前测试集
- 用剩余部分训练算法
- 在当前测试集上测试算法

(3) 记录每次得分及平均得分。

(4) 在上述过程中，每条数据只能在测试集中出现一次，以减少（但不能完全规避）运气成分。

> 书中同一章的代码，前后是紧密联系的，所以需要把它们放到一个IPython Notebook笔记本文件中，除非我告诉你不必这么做。

scikit-learn提供了几种交叉检验方法。有个辅助函数实现了上述交叉检验步骤，现在把它导进来。

```
from sklearn.cross_validation import cross_val_score
```

> cross_val_score默认使用Stratified K Fold方法切分数据集，它大体上保证切分后得到的子数据集中类别分布相同，以避免某些子数据集出现类别分布失衡的情况。这个默认做法很不错，现阶段就不再把它搞复杂了。

我们就来试试这个函数吧，把完整的数据集和类别值传给它。

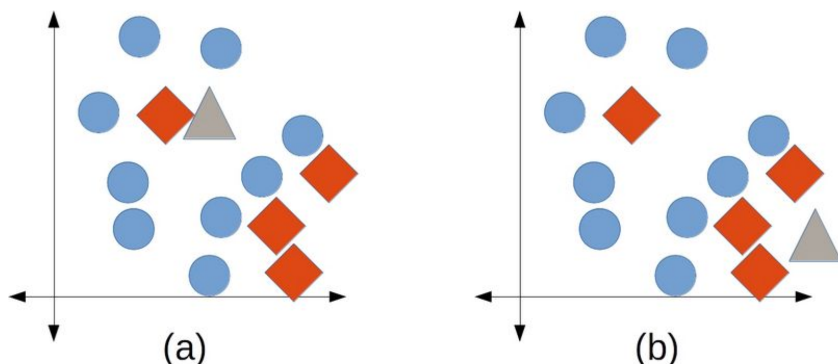
```
scores = cross_val_score(estimator, X, y, scoring='accuracy')
average_accuracy = np.mean(scores) * 100
print("The average accuracy is {0:.1f}%".format(average_accuracy))
```

哦，结果为82.3%，较之前稍微差点，但考虑到我们还没有尝试调整参数，这个结果还是相当不错的。下一节，我们就来研究怎么通过调整参数达到更理想的效果。

2.1.6　设置参数

几乎所有的数据挖掘算法都允许用户设置参数，这样做的好处是增强算法的泛化能力。但是，参数设置可是项技术活，选取好的参数值跟数据集的特征息息相关。

近邻算法有多个参数，最重要的是选取多少个近邻作为预测依据。scikit-learn管这个参数叫n_neighbors。下图给出两个极端的例子，n_neighbors过小时，分类结果容易受干扰，随机性很强。相反，如果n_neighbors过大，实际近邻的影响将削弱。



左图(a)中，我们通常希望把测试数据（三角形）归到圆形类别。然而，如果n_neighbors的值为1，由于三角形附近红色菱形（很可能是噪音）的存在，导致分类结果为菱形，虽然菱形集中在右下角区域。右图(b)中，我们希望将测试数据归到菱形类别。然而，如果n_neighbors值为7，三个最近的邻居（都是菱形）被四个圆形给击败了，三角形也因此被归到圆形类别。

如果想测试一系列n_neighbors的值，比如从1到20，可以重复进行多次实验，观察不同的参数值所带来的结果之间的差异。

```
avg_scores = []
all_scores = []
parameter_values = list(range(1, 21))  # Include 20
for n_neighbors in parameter_values:
    estimator = KNeighborsClassifier(n_neighbors=n_neighbors)
    scores = cross_val_score(estimator, X, y, scoring='accuracy')
```

把不同n_neighbors值的得分和平均分保存起来，留作分析用。

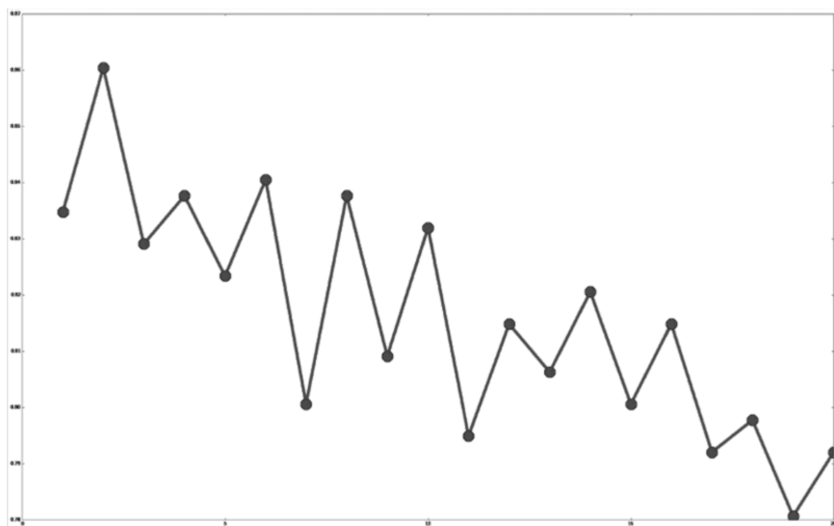
```
    avg_scores.append(np.mean(scores))
    all_scores.append(scores)
```

为了看起来更直观，我们可以用图表来表示n_neighbors的不同取值和分类正确率之间的关系。首先需要告诉IPython Notebook，我们要在笔记本中作图。

```
%matplotlib inline
```

然后，从matplotlib库导入pyplot，参数为近邻数和平均正确率。

```
from matplotlib import pyplot as plt plt.plot(parameter_values,
avg_scores, '-o')
```



从上图可以看到，虽然有很多曲折变化，但整体趋势是随着近邻数的增加，正确率不断下降。

2.2 流水线在预处理中的应用

现实中，物体不同特征的取值范围会非常广，它们的值域可能存在天壤之别。例如，测量动物的属性，会得到下面这样千差万别的特征值。

- 腿的数量：大多数动物有0到8条腿，但也有比这多得多的！
- 体重：从几微克到上百吨都有可能，有的蓝鲸重达190吨！
- 心脏数量：0到5之间，蚯蚓就有5颗心脏。

对于借助数学方法来比较特征的算法而言，它们很难理解特征在规模、范围和单位上的差异。如果我们在多种算法中使用上述特征，体重由于数值较大，可能都会是最显著的特征，但特征值大小实际上与该特征的分类效果没有任何关系。

不同特征的取值范围千差万别，常见的解决方法是对不同的特征进行规范化，使它们的特征值落在相同的值域或从属于某几个确定的类别，比如小、中和大。一旦解决这个问题，不同的特征类型对算法的影响将大大降低，分类正确率就能有大幅提升。

选择最具区分度的特征、创建新特征等都属于预处理的范畴。scikit-learn的预处理工具叫作转换器（Transformer），它接受原始数据集，返回转换后的数据集。除了处理数值型特征，转换器还能用来抽取特征。在这里，我们只看下对数值型特征的预处理方法。

2.2.1　预处理示例

为了讲解需要，先来对Ionosphere数据集做些破坏。虽然这里的麻烦是人为制造的，但是这些问题在很多真实数据集里都存在。首先，为了不破坏原来的数据集，我们为其创建一个副本。

```
X_broken = np.array(X)
```

接下来，我们就要捣乱了，将奇数列的特征值除以10。

```
X_broken[:,::2] /= 10
```

理论上讲，这样做对结果影响应该不大。毕竟，除以10之后，各个特征相差不大。主要的问题是，数值范围变了，奇数列的第二个特征要比偶数列的大。再次计算正确率看一下效果。

```
estimator = KNeighborsClassifier()
original_scores = cross_val_score(estimator, X, y,
  scoring='accuracy')
print("The original average accuracy for is
{0:.1f}%".format(np.mean(original_scores) * 100))
broken_scores = cross_val_score(estimator, X_broken, y,
  scoring='accuracy')
print("The 'broken' average accuracy for is
  {0:.1f}%".format(np.mean(broken_scores) * 100))
```

还记得吧，在原始数据集上的正确率为82.3%，这次跌至71.5%。把特征值转变到0到1之间就能解决这个问题。

2.2.2　标准预处理

我们接下来用MinMaxScaler类进行基于特征的规范化。在本章的笔记本文件中，接着之前的代码写，首先导入所需的类。

```

```

```
from sklearn.preprocessing import MinMaxScaler
```

这个类可以把每个特征的值域规范化为0到1之间。最小值用0代替，最大值用1代替，其余值介于两者之间。

接下来，对数据集X进行预处理。我们在预处理器MinMaxScaler上调用转换函数。有些转换器要求像训练分类器那样先进行训练，但是MinMaxScaler不需要，直接调用fit_transform()函数，即可完成训练和转换。

```
X_transformed = MinMaxScaler().fit_transform(X)
```

X_transformed与X行列数相等，为同型矩阵。然而，前者每列值的值域为0到1。

还有很多其他类似的规范化方法，对于其他类型的应用和特征类型会很有用。

- 为使每条数据各特征值的和为1，使用sklearn.preprocessing.Normalizer。
- 为使各特征的均值为0，方差为1，使用sklearn.preprocessing.StandardScaler，常用作规范化的基准。
- 为将数值型特征的二值化，使用sklearn.preprocessing.Binarizer，大于阈值的为1，反之为0。

后续章节，将会组合运用上述预处理方法及其他转换器对象。

2.2.3 组装起来

现在我们把前几节所讲的代码组合起来，创建一套完整的工作流，处理被破坏过的数据集。

```
X_transformed = MinMaxScaler().fit_transform(X_broken)
estimator = KNeighborsClassifier()
transformed_scores = cross_val_score(estimator, X_transformed, y,
  scoring='accuracy')
print("The average accuracy for is
```

```
{0:.1f}%".format(np.mean(transformed_scores) * 100))
```

正确率再次升到82.3%。MinMaxScaler将特征规范化到相同的值域，这样特征就不会仅仅因为值大而具备更强的区分度。简单总结下，异常值会影响近邻算法，不同算法对值域大小的敏感度不同。

2.3 流水线

随着实验的增加，操作的复杂程度也在提高。我们可能需要切分数据集，对特征进行二值化处理，以特征或数据集中的个体为基础规范化数据，除此之外还可能需要其他各种操作。

要跟踪记录所有这些操作可不容易，如果中间出点问题，先前实验的结果将很难再现。常见问题有落下步骤，数据转换错误，或进行了不必要的转换操作等。

另一个问题就是代码的先后顺序。上一节，我们创建了X_transformed数据集，然后创建了一个新的估计器用于交叉检验。如果有多个步骤，就需要跟踪代码中对数据集进行的每一步操作。

流水线结构就是用来解决这些问题的（当然不限于这些，下一章会讲到它在其他方面的应用）。流水线把这些步骤保存到数据挖掘的工作流中。之后你就可以用它们读入数据，做各种必要的预处理，然后给出预测结果。我们可以在cross_val_score等接收估计器的函数中使用流水线。创建流水线前，先导入Pipeline对象。

```
from sklearn.pipeline import Pipeline
```

流水线的输入为一连串的数据挖掘步骤，其中最后一步必须是估计器，前几步是转换器。输入的数据集经过转换器的处理后，输出的结果作为下一步的输入。最后，用位于流水线最后一步的估计器对数据进行分类。我们流水线分为两大步。

(1) 用MinMaxScaler将特征取值范围规范到0～1。

(2) 指定KNeighborsClassifier分类器。

每一步都用元组（‘名称’，步骤）来表示。现在来创建流水线。

```
scaling_pipeline = Pipeline([('scale', MinMaxScaler()),
                             ('predict', KNeighborsClassifier())])
```

流水线的核心是元素为元组的列表。第一个元组规范特征取值范围，第二个元组实现预测功能。我们把第一步叫作规范特征取值（scale），第二步叫作预测（predict），也可以用其他名字。元组的第二部分是实际的转换器对象或估计器对象。

流水线写好后，运行它很简单。使用先前用到的交叉检验代码看一下实际效果。

```
scores = cross_val_score(scaling_pipeline, X_broken, y,
scoring='accuracy')
print("The pipeline scored an average accuracy for is {0:.1f}%".
format(np.mean(transformed_scores) * 100))
```

运行结果跟之前一样（82.3%），表明我们这次用到的步骤跟之前相同。

后续章节会使用更高级的测试方法，而设置流水线就很有必要，因为它能确保代码的复杂程度不至于超出掌控范围。

2.4　小结

本章，我们用`scikit-learn`库提供的几个方法，创建了运行和评估数据挖掘模型的标准工作流。还介绍了近邻算法，`scikit-learn`将其封装为一个估计器，使用起来很简单：首先调用`fit`函数在训练集上进行训练，然后用`predict`函数在测试集上评估效果。

本章还通过解决不同特征值域影响分类效果的问题，讲解了预处理方法，主要用到了转换器对象和`MinMaxScaler`类。它们也有训练（`fit`）和转换方法，在转换阶段，接收数据集，返回处理过的数据集。

下一章，我们尝试把学到的这些概念应用到一个大一点的例子上，学着预测NBA（美国职业篮球联赛）比赛结果，其中使用的数据集可是真实的哦。

第 3 章　用决策树预测获胜球队

本章介绍另一种分类算法——决策树，用它预测NBA篮球赛的获胜球队。比起其他算法，决策树有很多优点，其中最主要的一个优点是决策过程是机器和人都能看懂的，我们使用机器学习到的模型就能完成预测任务。正如我们将在本章讲到的，决策树的另一个优点则是它能处理多种不同类型的特征。

本章主要内容有：

- 用pandas库加载、处理数据
- 决策树
- 随机森林
- 对真实数据集进行数据挖掘
- 创建新特征，用强有力的框架对其进行测试

3.1　加载数据集

本章将介绍怎样预测NBA获胜球队。如果你看过NBA，可能知道比赛中两支球队比分咬得很紧，难分胜负，有时最后一分钟才能定输赢，因此预测赢家很难。很多体育赛事都有类似的特点，预期的大赢家也许当天被另一支队伍给打败了。

以往很多对体育赛事预测的研究表明，正确率因体育赛事而异，其上限在70%~80%之间。体育赛事预测多采用数据挖掘或统计学方法。

3.1.1　采集数据

我们将使用NBA 2013—2014赛季的比赛数据。http://Basketball-Reference.com网站提供了NBA及其他赛事的大量资料和统计数据。请按以下方法下载数据。

(1) 在浏览器中打开http://www.basketball-reference.com/leagues/NBA_2014_games.html。

(2) 点击标题Regular Season旁边的Export按钮。

(3) 将文件下载到Data文件夹，记录文件的路径。

数据文件格式为CSV，包含了NBA常规赛季的1230场比赛。

CSV为简单的文本格式文件，每行为一条用逗号分隔的数据（文件格式的名字就是这么来的）。在记事本里输入内容，保存时使用.csv扩展名，也能生成CSV文件。只要能阅读文本文件的编辑器，就能打开CSV文件，也可以用Excel把它作为电子表格打开。

我们用pandas（Python Data Analysis的简写，意为Python数据分析）库加载这些数据，pandas在数据处理方面特别有用。Python内置了读写CSV文件的`csv`库。但是，考虑到后面创建新特征时还要用到pandas更强大的一些函数，所以我们干脆用pandas加载数据文件。



本章需要安装`pandas`。最简单的方法就是用`pip3`来安装，第1章中安装`scikitlearn`库时用的就是`pip3`。`pandas`的安装方法如下：

```
$pip3 install pandas
```

安装过程中若遇到任何困难，请访问http://pandas.pydata.org/getpandas.html，根据自己的系统，阅读相关安装指南。

3.1.2 用pandas加载数据集

pandas库是用来加载、管理和处理数据的。它在后台处理数据结构，支持诸计算均值等分析方法。

如果做过大量数据挖掘实验，就会发现自己翻来覆去地编写文件读取、特征抽取等函数。而这些函数每重新实现一次，都可能引入新错误。使用pandas等封装了很多功能的库，能有效减少反复实现上述函数所带来的工作量，并能保证代码的正确性。

本书后面会陆续介绍更多的数据挖掘案例，我们将大量使用pandas。

用`read_csv`函数就能加载数据集：

```
import pandas as pd
dataset = pd.read_csv(data_filename)
```

上述代码会加载数据集，将其保存到数据框（dataframe）中。数据框提供了一些非常好用的方法，后面会用到。我们来看看数据集是否有问题。输入以下代码，输出数据集的前5行：

```
dataset.ix[:5]
```

输出结果如下。

Out[47]:

Date	NaN	Visitor/Neutral	PTS	Home/Neutral	PTS	NaN	Notes
Tue Oct 29 2013	Box Score	Orlando Magic	87	Indiana Pacers	97	NaN	NaN
		Los Angeles Clippers	103	Los Angeles Lakers	116	NaN	NaN
		Chicago Bulls	95	Miami Heat	107	NaN	NaN
Wed Oct 30 2013	Box Score	Brooklyn Nets	94	Cleveland Cavaliers	98	NaN	NaN

从输出结果来看，这个数据集可以用，但存在几个小问题。下面我们就来修复这些问题。

3.1.3　数据集清洗

从上面的输出结果中，我们发现了以下几个问题。

- 日期是字符串格式，而不是日期对象。
- 第一行没有数据。
- 从视觉上检查结果，发现表头不完整或者不正确。

这些问题来自数据，我们可以改动数据本身，但是这样做的话，容易忘记之前做过哪些操作，落下步骤或是弄错哪一步，因而无法重现之前的结果。我们像前一章用流水线跟踪数据预处理流程那样，用pandas对原始数据进行预处理。

pandas.read_csv函数提供了可用来修复数据的参数，导入文件时指定这几个参数就好。导入后，我们还可以修改文件的头部，如下所示：

```
dataset = pd.read_csv(data_filename, parse_dates=["Date"],
```

```
  skiprows=[0,])
dataset.columns = ["Date", "Score Type", "Visitor Team",
  "VisitorPts", "Home Team", "HomePts", "OT?", "Notes"]
```

经过这些处理之后，结果会有很大改善，我们再来输出前5行看看：

```
dataset.ix[:5]
```

结果如下。

Out[48]:

	Date	Score Type	Visitor Team	VisitorPts	Home Team	HomePts	OT?	Notes
0	2013-10-29	Box Score	Orlando Magic	87	Indiana Pacers	97	NaN	NaN
1	2013-10-29	Box Score	Los Angeles Clippers	103	Los Angeles Lakers	116	NaN	NaN
2	2013-10-29	Box Score	Chicago Bulls	95	Miami Heat	107	NaN	NaN
3	2013-10-30	Box Score	Brooklyn Nets	94	Cleveland Cavaliers	98	NaN	NaN
4	2013-10-30	Box Score	Atlanta Hawks	109	Dallas Mavericks	118	NaN	NaN
5	2013-10-30	Box Score	Washington Wizards	102	Detroit Pistons	113	NaN	NaN

即使原始数据很规整，比如刚使用的这个，我们仍需要对其做些调整。其中一个原因是，文件可能来自不同的系统，由于存在兼容性问题，文件也许会发生变化。

既然数据已经准备好，在开始编写预测算法之前，我们先定下一个正确率作为基准。该基准任何算法都应该能达到。

每场比赛有两个队：主场队和客场队。最直接的方法就是拿几率作为基准，猜中的几率为50%。猜测任意一支球队获胜，都有一半胜算。

3.1.4 提取新特征

我们接下来通过组合和比较现有数据抽取特征。首先，确定类别值。在测试阶段，拿算法得到的分类结果与它对比，就能知道结果是否正确。类别可以有多种表示方法，我们这里用1表示主场队获胜，用0表示客场队获胜。对于篮球比赛而言，得分最多的队伍获胜。虽然数据集没有明确给出各球队的胜负情况，但是稍加计算就能得到。

找出主场获胜的球队：

```
dataset["HomeWin"] = dataset["VisitorPts"] < dataset["HomePts"]
```

我们把主场获胜球队的数据保存到NumPy数组里，稍后要用scikit-learn分类器对其进行处理。当前pandas和scikit-learn并没有进行整合，但是借助NumPy数组，它们配合地很好。我们用pandas抽取特征后再用scikit-learn抽取特征具体的值。

```
y_true = dataset["HomeWin"].values
```

上面的y_true数组保存的是类别数据，scikit-learn可直接读取该数组。

我们还可以创建一些特征用于数据挖掘。有时候，只要把原始数据丢给分类器就行了，但通常需要先抽取数值型或类别型特征。

首先，创建两个能帮助我们进行预测的特征，分别是这两支队伍上场比赛的胜负情况。赢得上场比赛，大致可以说明该球队水平较高。

遍历每一行数据，记录获胜球队。当到达一行新数据时，分别查看该行数据中的两支球队在各自的上一场比赛中有没有获胜的。

创建（默认）字典，存储球队上次比赛的结果。

```
from collections import defaultdict
won_last = defaultdict(int)
```

字典的键为球队，值为是否赢得上一场比赛。遍历所有行，在此过程中，更新每一行，为其增加两个特征值：两支球队在上场比赛有没有获胜。

```
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
    row["HomeLastWin"] = won_last[home_team]
    row["VisitorLastWin"] = won_last[visitor_team]
    dataset.ix[index] = row
```

请注意，上述代码假定数据集是按照时间顺序排列的。我们所使用的数据集是按这种顺序排列的，如果你的数据集不是这样，你需要把代码中的dataset.iterrows()替换为dataset.sort("Date").iterrows()。

用当前比赛（遍历到的那一行数据所表示的比赛）的结果更新两支球队上场比赛的获胜情况，以便下次再遍历到这两支球队时使用。代码如下：

```
    won_last[home_team] = row["HomeWin"]
    won_last[visitor_team] = not row["HomeWin"]
```

上述代码运行结束后，我们多了两个新特征：HomeLastWin和VisitorLastWin。我们再来看下数据集。这次只看前5条意义不大。只有一个球队参加过两场比赛后，我们才知道它在上场比赛表现如何。以下代码将输出本赛季第20～25场比赛：

```
dataset.ix[20:25]
```

输出结果如下：

Out[52]:

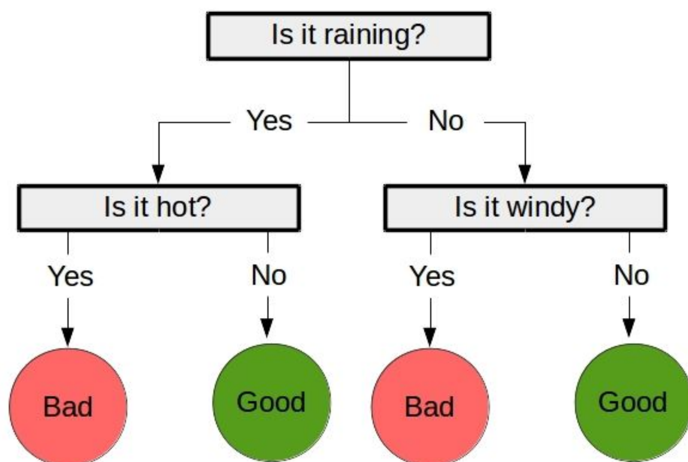
	Date	Score Type	Visitor Team	VisitorPts	Home Team	HomePts	OT?	Notes	HomeWin	HomeLastWin	VisitorLastWin
20	2013-11-01	Box Score	Milwaukee Bucks	105	Boston Celtics	98	NaN	NaN	False	False	False
21	2013-11-01	Box Score	Miami Heat	100	Brooklyn Nets	101	NaN	NaN	True	False	False
22	2013-11-01	Box Score	Cleveland Cavaliers	84	Charlotte Bobcats	90	NaN	NaN	True	False	True
23	2013-11-01	Box Score	Portland Trail Blazers	113	Denver Nuggets	98	NaN	NaN	False	False	False
24	2013-11-01	Box Score	Dallas Mavericks	105	Houston Rockets	113	NaN	NaN	True	True	True
25	2013-11-01	Box Score	San Antonio Spurs	91	Los Angeles Lakers	85	NaN	NaN	False	False	True

更换上述代码中的索引值，查看其他部分数据。别忘了，一共有1000多场比赛呢！

现在，每个队（包括上个赛季的冠军！）在数据集中第一次出现时，都假定它们在上场比赛中失败。其实可以用上一年的数据弥补缺失的信息，从而改进这个特征，但这里就先做简单处理了。

3.2　决策树

决策树是一种有监督的机器学习算法，它看起来就像是由一系列节点组成的流程图，其中位于上层节点的值决定下一步走向哪个节点。



跟大多数分类算法一样，决策树也分为两大步骤。

- 首先是训练阶段，用训练数据构造一棵树。上一章的近邻算法没有训练阶段，但是决策树需要。从这个意义上说，近邻算法是一种惰性算法，在用它进行分类时，它才开始干活。相反，决策树跟大多数机器学习方法类似，是一种积极学习的算法，在训练阶段完成模型的创建。

- 其次是预测阶段，用训练好的决策树预测新数据的类别。以上图为例，`["is raining", "very windy"]`的预测结果为“Bad”（坏天气）。

创建决策树的算法有多种，大都通过迭代生成一棵树。它们从根节点开始，选取最佳特征，用于第一个决策，到达下一个节点，选择下一个最佳特征，以此类推。当发现无法从增加树的层级中获得更多信息时，算法启动退出机制。

`scikit-learn`库实现了分类回归树（Classification and Regression Trees，CART）算法并将其作为生成决策树的默认算法，它支持连续型特征和类别型特征。

3.2.1 决策树中的参数

退出准则是决策树的一个重要特性。构建决策树时，最后几步决策仅依赖于少数个体，随意性大。使用特定节点作出推测容易导致过拟合训练数据，而使用退出准则可以防止决策精度过高。

除了设定退出准则外，也可以先创建一棵完整的树，再对其进行修剪，去掉对整个过程没有提供太多信息的节点。这个过程叫作剪枝（pruning）。

scikit-learn库实现的决策树算法给出了退出方法，使用下面这两个选项就可以达到目的。

- min_samples_split：指定创建一个新节点至少需要的个体数量。
- min_samples_leaf：指定为了保留节点，每个节点至少应该包含的个体数量。

第一个参数控制着决策节点的创建，第二个参数决定着决策节点能否被保留。

决策树的另一个参数是创建决策的标准，常用的有以下两个。

- 基尼不纯度（Gini impurity）：用于衡量决策节点错误预测新个体类别的比例。
- 信息增益（Information gain）：用信息论中的熵来表示决策节点提供多少新信息。

3.2.2　使用决策树

从scikit-learn库中导入DecisionTreeClassifier类，用它创建决策树。

```
from sklearn.tree import DecisionTreeClassifier
clf = DecisionTreeClassifier(random_state=14)
```

> 我们再次设定random_state的值为14。本书中凡是用到random_state的地方，我们都用该值。使用相同的随机种子（random seed），能够保证几次实验结果相同。然而，在以后自己的实验中，为保证算法的性能不是与特定的随机状态值相关，在前后几次实验中，需使用不同的随机状态。

现在我们从pandas数据框中抽取数据，以便用scikit-learn分类器处理。指定需要的列，使用数据框的values属性，就能获取到每支

球队的上一场比赛结果。

```
X_previouswins = dataset[["HomeLastWin", "VisitorLastWin"]].values
```

跟第2章的近邻算法类似，决策树也是一种估计器，因此它同样有fit和predict方法。我们仍然可以用cross_val_score方法来求得交叉检验的平均正确率：

```
scores = cross_val_score(clf, X_previouswins, y_true,
scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

正确率为56.1%，比起随机预测来要更准确！我们应该可以做得更好。从数据集中构建有效特征（Feature Engineering，特征工程）是数据挖掘的难点所在，好的特征直接关系到结果的正确率——甚至比选择合适的算法更重要！

3.3 NBA比赛结果预测

尝试使用不同的特征，我们应该能做得更好。cross_val_score方法可用来测试模型的正确率。有了它，我们就可以尝试其他特征的分类效果。

好多潜在特征都可以拿来用。就我们这个挖掘任务而言，具体怎么选择特征呢？我们尝试问自己以下两个问题。

- 一般而言，什么样的球队水平更高？
- 两支球队上一次相遇时，谁是赢家？

我们还将加入新球队的数据，以检测算法是否能得到一个用来判断不同球队比赛情况的模型。

组装起来

对于上面第一个特征，我们创建一个叫作“主场队是否通常比对手水平高”的特征，并使用2013赛季的战绩作为特征取值来源。如果一支球队在2013赛季排名在对手前面，我们就认为它的水平更高。

战绩数据下载方法如下。

(1) 在浏览器中打开http://www.basketball-reference.com/leagues/NBA_2013_standings.html。

(2) 找到Expanded Standings部分，该部分包括所有球队的数据。

(3) 点击Export链接。

(4) 将数据保存到数据文件夹中。

回到IPython Notebook笔记本文件，在新格子里输入以下代码。请注意将战绩文件保存到`data_folder`变量指定的目录下。

```
standings_filename = os.path.join(data_folder,
"leagues_NBA_2013_standings_expanded-standings.csv")
standings = pd.read_csv(standings_filename, skiprows=[0,1])
```

在笔记本新格子里输入`standings`并运行，查看战绩。

```
standings
```

输出如下。

Out[57]:

	Rk	Team	Overall	Home	Road	E	W	A	C	SE	...	Post	≤3	≥10	Oct	Nov	Dec	Jan	Feb	Mar	Apr
0	1	Miami Heat	66-16	37-4	29-12	41-11	25-5	14-4	12-6	15-1	...	30-2	9-3	39-8	1-0	10-3	10-5	8-5	12-1	17-1	8-1
1	2	Oklahoma City Thunder	60-22	34-7	26-15	21-9	39-13	7-3	8-2	6-4	...	21-8	3-6	44-6	NaN	13-4	11-2	11-5	7-4	12-5	6-2
2	3	San Antonio Spurs	58-24	35-6	23-18	25-5	33-19	8-2	9-1	8-2	...	16-12	9-5	31-10	1-0	12-4	12-4	12-3	8-3	10-4	3-6
3	4	Denver Nuggets	57-25	38-3	19-22	19-11	38-14	5-5	10-0	4-6	...	24-4	11-7	28-8	0-1	8-8	9-6	12-3	8-4	13-2	7-1
4	5	Los Angeles Clippers	56-26	32-9	24-17	21-9	35-17	7-3	8-2	6-4	...	17-9	3-5	38-12	1-0	8-6	16-0	9-7	8-5	7-7	7-1
5	6	Memphis Grizzlies	56-26	32-9	24-17	22-8	34-18	8-2	8-2	6-4	...	23-8	6-4	28-9	0-1	12-1	7-7	10-7	9-2	11-6	7-2
6	7	New York Knicks	54-28	31-10	23-18	37-15	17-13	10-6	12-6	15-3	...	22-10	7-5	31-12	NaN	11-4	10-5	7-6	6-5	12-6	8-2
7	8	Brooklyn Nets	49-33	26-15	23-18	36-16	13-17	11-5	13-5	12-6	...	18-11	9-4	23-17	NaN	11-4	5-11	11-4	7-5	8-7	7-2
8	9	Indiana Pacers	49-32	30-11	19-21	31-20	18-12	6-11	13-3	12-6	...	17-11	4-9	27-14	1-0	7-8	10-5	9-6	9-3	11-5	2-5
9	10	Golden State Warriors	47-35	28-13	19-22	19-11	28-24	7-3	5-5	7-3	...	17-13	5-3	20-18	1-0	8-6	12-4	8-7	4-8	9-7	5-3

接下来，创建一个新特征，创建过程与上个特征类似。遍历每一行，查找主场队和客场队两支球队的战绩。代码如下：

```
dataset["HomeTeamRanksHigher"] = 0
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
```

有些球队2014赛季改名了，名字不同但其实还是同一支球队。类似情况在整合不同的数据集时经常遇到！所以在查找球队时，需要把它换成原来的名字，以确保正确找到该球队先前的排名。

```
    if home_team == "New Orleans Pelicans":
        home_team = "New Orleans Hornets"
    elif visitor_team == "New Orleans Pelicans":
        visitor_team = "New Orleans Hornets"
```

现在就能得到两支球队的排名，比较它们的排名，更新特征值。

```
    home_rank = standings[standings["Team"] ==
          home_team]["Rk"].values[0]
          visitor_rank = standings[standings["Team"] ==
          visitor_team]["Rk"].values[0]
          row["HomeTeamRanksHigher"] = int(home_rank > visitor_rank)
          dataset.ix[index] = row
```

接下来，用cross_val_score函数测试结果。首先，从数据集中抽取所需要的部分。

```
X_homehigher = dataset[["HomeLastWin", "VisitorLastWin",
  "HomeTeamRanksHigher"]].values
```

然后，创建DecisionTreeClassifier分类器，进行交叉检验，求得正确率。

```
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_homehigher, y_true,
  scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

现在的正确率是60.3%——比我们之前的结果要好。还能再提高吗？

接下来，我们来统计两支球队上场比赛的情况，作为另一个特征。虽然球队排名有助于预测（排名靠前的胜算更大），但有时排名靠后的球队反而能战胜排名靠前的。原因有很多。例如，排名靠后的球队某些打法恰好能击中强者的软肋。该特征的创建方法与前一个特征类似，首先创建字典，保存上场比赛的获胜队伍，在数据框中建立新特征。代码如下：

```
last_match_winner = defaultdict(int)
dataset["HomeTeamWonLast"] = 0
```

然后，遍历每条数据，取到每场赛事的两支参赛队伍。

```
for index, row in dataset.iterrows():
    home_team = row["Home Team"]
    visitor_team = row["Visitor Team"]
```

不用考虑哪支球队是主场作战，我们想看一下这两支球队在上一场比赛中到底谁是赢家。因此，按照英文字母表顺序对球队名字进行排序，确保两支球队无论主客场作战，都使用相同的键。

```
    teams = tuple(sorted([home_team, visitor_team]))
```

通过查找字典，找到两支球队上次比赛的赢家。然后，更新数据框中这条数据。

```
    row["HomeTeamWonLast"] = 1 if last_match_winner[teams] ==
      row["Home Team"] else 0
      dataset.ix[index] = row
```

最后，更新last_match_winner字典，值为两支球队在当前场次比赛中的胜出者，两支球队再相逢时可将其作为参考。

```
    winner = row["Home Team"] if row["HomeWin"] else row
     ["Visitor Team"]
    last_match_winner[teams] = winner
```

下面，用新抽取的两个特征创建数据集。观察不同特征组合的分类效果。代码如下：

```
X_lastwinner = dataset[["HomeTeamRanksHigher", "HomeTeam
  WonLast"]].values
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_lastwinner, y_true,
  scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

正确率为60.6%。结果越来越好了。

最后我们来看一下，决策树在训练数据量很大的情况下，能否得到有效的分类模型。我们将会为决策树添加球队，以检测它是否能整合新增的信息。

虽然决策树能够处理特征值为类别型的数据，但scikit-learn库所实现的决策树算法要求先对这类特征进行处理。用LabelEncoder转换器就能把字符串类型的球队名转化为整型。代码如下：

```
from sklearn.preprocessing import LabelEncoder
encoding = LabelEncoder()
```

将主场球队名称转化为整型：

```
encoding.fit(dataset["Home Team"].values)
```

接下来，抽取所有比赛的主客场球队的球队名（已转化为数值型）并将其组合（在NumPy中叫作“stacking”，是向量组合的意思）起来，形成一个矩阵。代码如下：

```
home_teams = encoding.transform(dataset["Home Team"].values)
visitor_teams = encoding.transform(dataset["Visitor Team"].values)
X_teams = np.vstack([home_teams, visitor_teams]).T
```

决策树可以用这些特征值进行训练，但DecisionTreeClassifier仍把它们当作连续型特征。例如，编号从0到16的17支球队，算法会认为球队1和2相似，而球队4和10不同。但其实这没意义，对于两支球队而言，它们要么是同一支球队，要么不同，没有中间状态！

为了消除这种和实际情况不一致的现象，我们可以使用OneHotEncoder转换器把这些整数转换为二进制数字。每个特征用一个二进制数字1来表示。例如，LabelEncoder为芝加哥公牛队分配的数值是7，那么OneHotEncoder为它分配的二进制数字的第七位就是1，其余队伍的第七位就是0。每个可能的特征值都这样处理，而数据集会变得很大。代码如下：

> 1有多少个特征，二进制数字就有多少位。——译者注

```

```

```
from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder()
```

在相同的数据集上进行预处理和训练操作，将结果保存起来备用。

```
X_teams_expanded = onehot.fit_transform(X_teams).todense()
```

接着，像之前那样在新数据集上调用决策树分类器。

```
clf = DecisionTreeClassifier(random_state=14)
scores = cross_val_score(clf, X_teams_expanded, y_true,
  scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

正确率为60%，比基准值要高，但是没有之前的效果好。原因可能在于特征数增加后，决策树处理不当。鉴于此，我们尝试修改算法，看看会不会起作用。数据挖掘有时就是不断尝试新算法、使用新特征这样一个过程。

3.4　随机森林

一棵决策树可以学到很复杂的规则。然而，很可能会导致过拟合问题——学到的规则只适用于训练集。解决方法之一就是调整决策树算法，限制它所学到的规则的数量。例如，把决策树的深度限制在三层，只让它学习从全局角度拆分数据集的最佳规则，不让它学习适用面很窄的特定规则，这些规则会将数据集进一步拆分为更加细致的群组。使用这种折中方案得到的决策树泛化能力强，但整体表现稍弱。

为了弥补上述方法的不足，我们可以创建多棵决策树，用它们分别进行预测，再根据少数服从多数的原则从多个预测结果中选择最终预测结果。这正是随机森林的工作原理。

但上述过程有两个问题。一是创建的多棵决策树在很大程度上是相同的——每次使用相同的输入，将得到相同的输出。我们只有一个训练集，如果尝试创建多棵决策树，它们的输入就可能相同（因此输出也相同）。解决方法是每次随机从数据集中选取一部分数据用作训练集。这个过程叫作装袋（bagging）。

第二个问题是用于前几个决策节点的特征非常突出。即使我们随机选

取部分数据用作训练集，创建的决策树相似性仍旧很大。解决方法是，随机选取部分特征作为决策依据。

然后，使用随机从数据集中选取的数据和（几乎是）随机选取的特征，创建多棵决策树。这就是随机森林，虽然看上去不是那么直观，但这种算法在很多数据集上效果很好。

3.4.1　决策树的集成效果如何

随机森林算法内在的随机性让人感觉算法的好坏全靠运气。然而，通过对多棵几乎是随机创建的决策树的预测结果取均值，就能降低预测结果的不一致性。我们用方差来表示这种不一致。

方差是由训练集的变化引起的。决策树这类方差大的算法极易受到训练集变化的影响，从而产生过拟合问题。

> 对比来说，偏误（bias）是由算法中的假设引起的，而与数据集没有关系。比如，算法错误地假定所有特征呈正态分布，就会导致较高的误差。通过分析分类器的数据模型和实际数据集的匹配情况，就能降低偏误问题的负面影响。

对随机森林中大量决策树的预测结果取均值，能有效降低方差，这样得到的预测模型的总体正确率更高。

一般而言，决策树集成做出了如下假设：预测过程的误差具有随机性，且因分类器而异。因此，使用由多个模型得到的预测结果的均值，能够消除随机误差的影响——只保留正确的预测结果。本书中会介绍更多用集成方法消除误差的例子。

3.4.2　随机森林算法的参数

`scikit-learn`库中的`RandomForestClassifier`就是对随机森林算法的实现，它提供了一系列参数。因为它使用了`DecisionTreeClassifier`的大量实例，所以它俩的很多参数是一致的，比如决策标准（基尼不纯度/信息增益）、`max_features`和`min_samples_split`。

当然，集成过程还引入了一些新参数。

- `n_estimators`：用来指定创建决策树的数量。该值越高，所

花时间越长，正确率（可能）也越高。

- oob_score：如果设置为真，测试时将不使用训练模型时用过的数据。

- n_jobs：采用并行计算方法训练决策树时所用到的内核数量。

scikit-learn库提供了用于并行计算的Joblib库。n_jobs指定所用的内核数。默认使用1个内核——如果CPU是多核的，可以多用几个，或者将其设置为-1，开动全部马力。

3.4.3 使用随机森林算法

scikit-learn库实现的随机森林算法使用估计器接口，用交叉检验方法调用它即可，代码跟之前大同小异。

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier(random_state=14)
scores = cross_val_score(clf, X_teams, y_true, scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

只是靠更换分类器，正确率就提升了0.6%，达到60.6%。

随机森林使用不同的特征子集进行学习，应该比普通的决策树更为高效。下面来看一下多用几个特征效果如何。

```
X_all = np.hstack([X_home_higher, X_teams])
clf = RandomForestClassifier(random_state=14)
scores = cross_val_score(clf, X_all, y_true, scoring='accuracy')
print("Accuracy: {0:.1f}%".format(np.mean(scores) * 100))
```

正确率为61.1%——又有所提升！可以使用GridSearchCV类搜索最佳参数，代码如下：

```
parameter_space = {
  "max_features": [2, 10, 'auto'],
  "n_estimators": [100,],
  "criterion": ["gini", "entropy"],
  "min_samples_leaf": [2, 4, 6],
}
clf = RandomForestClassifier(random_state=14)
grid = GridSearchCV(clf, parameter_space)
grid.fit(X_all, y_true)
```

```
print("Accuracy: {0:.1f}%".format(grid.best_score_ * 100))
```

这次正确率提升较大，达到了64.2%！

输出用网格搜索找到的最佳模型，查看都使用了哪些参数。代码如下：

```
print(grid.best_estimator_)
```

下面的代码将给出正确率最高的模型所用到的参数。

```
RandomForestClassifier(bootstrap=True, compute_importances=None,
  criterion='entropy', max_depth=None, max_features=2,
    max_leaf_nodes=None, min_density=None, min_samples_leaf=6,
      min_samples_split=2, n_estimators=100, n_jobs=1,
        oob_score=False, random_state=14, verbose=0)
```

3.4.4 创建新特征

从上述几个例子中，我们看到改变特征对算法的表现有很大影响。经过小规模测试发现，仅仅是因为选用不同的特征，正确率竟然提升了10%。

用pandas提供的函数创建特征。代码如下：

```
dataset["New Feature"] = feature_creator()
```

`feature_creator`函数返回数据集中每条数据的各个特征值。常用数据集作为参数。

```
dataset["New Feature"] = feature_creator(dataset)
```

最直接的做法是一开始为新特征设置默认的值，比如0。如下所示：

```
dataset["My New Feature"] = 0
```

接下来，遍历数据集，计算所需特征。本章曾多次用下面这种形式创

建新特征。

```
for index, row in dataset.iterrows():
  home_team = row["Home Team"]
  visitor_team = row["Visitor Team"]
  # Some calculation here to alter row
  dataset.ix[index] = row
```

请注意，上面这种遍历方法效率不高。如果你要用的话，请一次性处理所有特征。常用的“最佳做法”就是每条数据最好只处理一次。

你可以创建下述特征并看一下效果。

- 球队上次打比赛距今有多长时间？短期内连续作战，容易导致球员疲劳。
- 两支球队过去五场比赛结果如何？这两个数据要比HomeLastWin和VisitorLastWin更能反映球队的真实水平（抽取特征方法类似）。
- 球队是不是跟某支特定球队打比赛时发挥得更好？例如，球队在某个体育场里打比赛，即使是客场作战也能发挥得很好。

如果抽取上面这些特征遇到困难，请参考pandas的文档http://pandas.pydata.org/pandas-docs/stable/。此外，还可以尝试从Stack Overflow等社区寻求帮助。

更极致的做法是，借助球员数据来分析每个队的实力以预测输赢。其实，赌徒和体育博彩机构每天都在用这些复杂的特征预测赛事结果来牟利。

3.5 小结

本章使用scikit-learn库的另一个分类器DecisionTreeClassifier，并介绍了如何用pandas处理数据。我们分析了真实的NBA赛事的比赛结果数据，创建新特征用于分类，并在这个过程中发现即使是规整、干净的数据也可能存在一些小问题。

我们发现好的特征对提升正确率很有帮助，还使用了一种集成算法——随机森林，进一步提升正确率。

下一章将会扩展在第1章使用的亲和性分析算法，用来发现相似的电影。我们还将学到如何用算法解决排序问题，以及如何提升数据挖掘算法的可扩展性。

第 4 章　用亲和性分析方法推荐电影

本章学习如何用亲和性分析方法找出在什么情况下两个对象经常一起出现。通俗来讲，这也叫“购物篮分析”，因为曾有人用它找出哪些商品经常一起出售。

第3章关注的对象为球队，并用特征描述球队。本章所用到的电影评分数据有所不同，我们所关注的对象（电影）埋在数据中。本章数据挖掘任务的目标是找出对象同时出现的情况，也就是寻找用户同时喜欢几部电影的情况。

本章主要涉及以下几个概念。

- 亲和性分析
- 用Apriori算法挖掘关联特征
- 电影推荐
- 数据稀疏问题

4.1　亲和性分析

亲和性分析用来找出两个对象共同出现的情况。而前几章，我们关注的是同种对象之间的相似度。亲和性分析所用的数据通常为类似于交易信息的数据。从直观上来看，这些数据就像是商店的交易数据——从中能看出哪些商品是顾客一起购买的。

然而，亲和性分析方法的应用场景有很多，比如：

- 欺诈检测
- 顾客区分
- 软件优化
- 产品推荐

亲和性分析比分类更具探索性，因为通常我们无法拿到像在很多分类任务中所用的那样完整的数据集。例如，在电影推荐任务中，我们拿到的是不同用户对不同电影的评价。但是，每个用户不可能评价过所有电影，这就给亲和性分析带来一个不容忽视的大难题。如果用户没有评价过一部电影，是因为他们不喜欢这部电影（据此就不推荐给他们），还是因为他们出于别的原因还没有评价？

本章不对上述问题做出解答，但是我们要思考数据集中类似这样的潜在问题该怎么解决。这些思考有助于提升推荐算法的准确性。

4.1.1　亲和性分析算法

我们在第1章介绍了一种基础的亲和性分析算法，尝试了所有可能的规则组合，计算了每条规则的置信度和支持度，并根据这两个标准进行排序，选取最佳规则。

然而，这个方法效率不高。好在第1章所使用的数据集中，每条交易数据只涉及五种商品。但现实并非如此，即使是再不起眼的小卖铺出售的商品也达上百种之多，网店更是有成千上万种商品（甚至几百万种！）。如果规则生成方法像第1章那样过于简单，计算这些规则所需要的时间复杂度将呈指数级增长。随着商品数量的增加，计算所有规则所需的时间增长得很快。更具体地说，所有可能的规则数量是$2n-1$。数据集有5个特征，可能的规则就有31条；有10个特征，可能的规则就有1023条；仅仅有100个特征，规则数就能达到30位数字。即使计算能力大幅提升也未必能赶上在线商品的增长速度。因此，与其跟计算机过不去，不如寻找更加聪明的算法。

Apriori算法可以说是经典的亲和性分析算法。它只从数据集中频繁出现的商品中选取共同出现的商品组成频繁项集（frequent itemset），避免了上述复杂度呈指数级增长的问题。一旦找到频繁项集，生成关联规则就很容易了。

Apriori算法背后的原理简洁却不失巧妙。首先，确保了规则在数据集中有足够的支持度。Apriori算法的一个重要参数就是最小支持度。比如，要生成包含商品A、B的频繁项集（A, B），要求支持度至少为30，那么A和B都必须至少在数据集中出现30次。更大的频繁项集也要遵守该项约定，比如要生成频繁项集（A, B, C, D），那么子集(A, B, C)必须是频繁项集（当然D自己也要满足最小支持度标准）。

生成频繁项集后，将不再考虑其他可能的却不够频繁的项集（这样的集合有很多），从而大大减少测试新规则所需的时间。

其他亲和性分析算法有Eclat和频繁项集挖掘算法（FP-growth）。从数据挖掘角度看，这些算法比起基础的Apriori算法有很多改进，性能也有进一步提升。接下来，先来看一下最基础的Apriori算法。

4.1.2　选择参数

挖掘亲和性分析所用的关联规则之前，我们先用Apriori算法生成频繁项集。接着，通过检测频繁项集中前提和结论的组合，生成关联规则（例如，如果用户喜欢电影X，那么他很可能喜欢电影Y）。

第一个阶段，需要为Apriori算法指定一个项集要成为频繁项集所需的最小支持度。任何小于最小支持度的项集将不再考虑。如果最小支持度值过小，Apriori算法要检测大量的项集，会拖慢的运行速度；最小支持度值过大的话，则只有很少的频繁项集。

找出频繁项集后，在第二个阶段，根据置信度选取关联规则。可以设定最小置信度，返回一部分规则，或者返回所有规则，让用户自己选。

本章，我们设定最小置信度，只返回高于它的规则。置信度过低将会导致规则支持度高，正确率低；置信度过高，导致正确率高，但是返回的规则少。

4.2　电影推荐问题

产品推荐技术是门大生意。网店经常用它向潜在用户推荐他们可能购买的产品。好的推荐算法能带来更高的销售业绩。每年有几百万乃至几千万用户进行网购，向他们推荐更多的商品，潜在收益着实可观。

产品推荐问题被人们研究了多年，但它一直不温不火，直到2007年到2009年间，Netflix公司推出数据建模大赛，并设立Netflix Prize奖项之后，才得到迅猛发展。该竞赛意在寻找比Netflix公司所使用的预测用户为电影打分的系统更准确的解决方案。最后获奖队伍以比现有系统高10个百分点的优势胜出。虽然这个改进看起来不是很大，但是Netflix公司却能借助它实现更精准的电影推荐服务，从而多赚上百万美元。

4.2.1　获取数据集

自打Netflix Prize奖项设立以来，美国明尼苏达大学的Grouplens研究

团队公开了一系列用于测试推荐算法的数据集。其中，就包括几个大小不同的电影评分数据集，分别有10万、100万和1000万条电影评分数据。

数据集下载地址为http://grouplens.org/datasets/movielens/。本章将使用包含10万条数据的MovieLens数据集。下载数据集，解压到你的Data文件夹。启动IPython Notebook笔记本，输入以下代码。

```
import os
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "ml-100k")
ratings_filename = os.path.join(data_folder, "u.data")
```

确保ratings_filename指向解压后得到的文件夹中的u.data文件。

4.2.2　用pandas加载数据

MovieLens数据集非常规整，但是有几点跟pandas.read_csv方法的默认设置有出入，所以要调整参数设置。第一个问题是数据集每行的几个数据之间用制表符而不是逗号分隔。其次，没有表头，这表示数据集的第一行就是数据部分，我们需要手动为各列添加名称。

加载数据集时，把分隔符设置为制表符，告诉pandas不要把第一行作为表头（header=None），设置好各列的名称。代码如下：

```
all_ratings = pd.read_csv(ratings_filename, delimiter="\t",
    header=None, names = ["UserID", "MovieID", "Rating", "Datetime"])
```

虽然本章用不到，还是稍微提一下，你可以用下面的代码解析时间戳数据。

```
all_ratings["Datetime"] = pd.to_datetime(all_ratings['Datetime'],
  unit='s')
```

运行下面的代码，看一下前五条记录。

```
all_ratings[:5]
```

输出结果如下。

		[illegible]		
1997-12-04 15:55:49				
1998-04-04 19:22:22				
1997-11-07 07:18:36				
1997-11-27 05:02:03				
1998-02-02 05:33:16				

4.2.3　稀疏数据格式

这是一个稀疏数据集，我们可以将每一行想象成巨大特征矩阵的一个格子，前几章用到过这种矩阵。在这个矩阵中，每一行表示一个用户，每一列为一部电影。第一列为每一个用户给第一部电影打的分数，第二列为每一个用户给第二部电影打的分数，以此类推。

数据集中有1000名用户和1700部电影，这就意味着整个矩阵很大。将矩阵读到内存中及在它基础上进行计算可能存在难度。然而，这个矩阵的很多格子都是空的，也就是对大部分用户来说，他们只给少数几部电影打过分。比如用户#213没有为电影#675打过分，大部分用户没有为大部分电影打过分。

用上述图表中的格式也能表示矩阵，且更为紧凑。序号为0的那一行表示，用户#196在1997年12月4日为电影#242打了3分（满分是5分）。

任何没有出现在数据集中的用户和电影组合表示它们实际上是不存在的。这比起在内存中保存大量的0，节省了很多空间。这种格式叫作稀疏矩阵（sparse matrix）。根据经验来说，如果数据集中60%或以上的数据为0，就应该考虑使用稀疏矩阵，从而节省不少空间。

在对稀疏矩阵进行计算时，我们关注的通常不是那些不存在的数据，不会去比较众多的0值，相反我们关注的是现有数据，并对它们进行比较。

4.3　Apriori算法的实现

本章数据挖掘的目标是生成如下形式的规则：如果用户喜欢某些电影，那么他们也会喜欢这部电影。作为对上述规则的扩展，我们还将讨论喜欢某几部电影的用户，是否喜欢另一部电影。

要解决以上问题，首先要确定用户是不是喜欢某一部电影。为此创建新特征`Favorable`，若用户喜欢该电影，值为`True`。

```
all_ratings["Favorable"] = all_ratings["Rating"] > 3
```

我们在数据集中看一下这个新特征。

```
all_ratings[10:15]
```

[illegible]
2007-11-12 22:07:14
2007-11-17 15:38:45
2007-10-05 09:05:40
2008-03-27 21:59:54
2008-02-21 23:40:57

从数据集中选取一部分数据用作训练集，这能有效减少搜索空间，提升Apriori算法的速度。我们取前200名用户的打分数据。

```
ratings = all_ratings[all_ratings['UserID'].isin(range(200))]
```

接下来，新建一个数据集，只包括用户喜欢某部电影的数据行。

```
favorable_ratings = ratings[ratings["Favorable"]]
```

在生成项集时，需要搜索用户喜欢的电影。因此，接下来，我们需要知道每个用户各喜欢哪些电影，按照User ID进行分组，并遍历每个用户看过的每一部电影。

```
favorable_reviews_by_users = dict((k, frozenset(v.values))
                                  for k, v in favorable_ratings
                                  groupby("UserID")["MovieID"])
```

上面的代码把`v.values`存储为`frozenset`，便于快速判断用户是否为某部电影打过分。对于这种操作，集合比列表速度快，在后面代码中还会用到。

最后，创建一个数据框，以便了解每部电影的影迷数量。

```
num_favorable_by_movie = ratings[["MovieID", "Favorable"]].
  groupby("MovieID").sum()
```

用以下代码查看最受欢迎的五部电影。

```
num_favorable_by_movie.sort("Favorable", ascending=False)[:5]
```

输出结果如下：

MovieID	Favorable
50	100
100	89
258	83
181	79
174	74

4.3.1 Apriori算法

Apriori算法是亲和性分析的一部分，专门用于查找数据集中的频繁项集。基本流程是从前一步找到的频繁项集中找到新的备选集合，接着检测备选集合的频繁程度是否够高，然后算法像下面这样进行迭代。

(1) 把各项目放到只包含自己的项集中，生成最初的频繁项集。只使用达到最小支持度的项目。

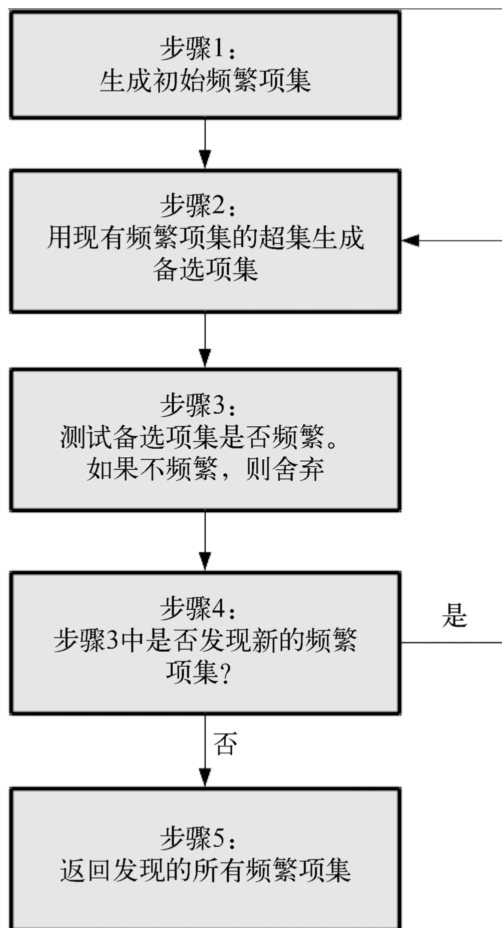
(2) 查找现有频繁项集的超集，发现新的频繁项集，并用其生成新的备选项集。

(3) 测试新生成的备选项集的频繁程度，如果不够频繁，则舍弃。如果没有新的频繁项集，就跳到最后一步。

(4) 存储新发现的频繁项集，跳到步骤(2)。

(5) 返回发现的所有频繁项集。

整个过程表示如下。



4.3.2　实现

Apriori算法第一次迭代时，新发现的项集长度为2，它们是步骤(1)中创建的项集的超集。第二次迭代（经过步骤(4)）中，新发现的项集长度为3。这有助于我们快速识别步骤(2)所需的项集。

我们把发现的频繁项集保存到以项集长度为键的字典中，便于根据长度查找，这样就可以找到最新发现的频繁项集。下面的代码初始化一个字典。

```
frequent_itemsets = {}
```

我们还需要确定项集要成为频繁项集所需的最小支持度。这个值需要根据数据集的具体情况来设定，可自行尝试其他值，建议每次只改动10个百分点，即使这样你可能也会发现算法运行时间变动很大！下面，设置最小支持度。

```
min_support = 50
```

我们先来实现Apriori算法的第一步，为每一部电影生成只包含它自己的项集，检测它是否够频繁。电影编号使用frozenset，后面要用到集合操作。此外，它们也可以用作字典的键（普通集合不可以）。代码如下：

```
frequent_itemsets[1] = dict((frozenset((movie_id,)),
                                  row["Favorable"])
                                  for movie_id, row in num_favorable_
                                    by_movie.iterrows()
                                  if row["Favorable"] > min_support)
```

接着，用一个函数来实现步骤(2)和(3)，它接收新发现的频繁项集，创建超集，检测频繁程度。下面为函数声明及字典初始化代码。

```
from collections import defaultdict
def find_frequent_itemsets(favorable_reviews_by_users, k_1_itemsets,
min_support):
    counts = defaultdict(int)
```

经验告诉我们，要尽量减少遍历数据的次数，所以每次调用函数时，再遍历数据。这样做效果不是很明显（因为数据集相对较小），但是数据集更大的情况下，就很有必要。我们来遍历所有用户和他们的打分数据。

```
    for user, reviews in favorable_reviews_by_users.items():
```

接着，遍历前面找出的项集，判断它们是否是当前评分项集的子集。如果是，表明用户已经为子集中的电影打过分。代码如下：

```
        for itemset in k_1_itemsets:
            if itemset.issubset(reviews):
```

接下来，遍历用户打过分却没有出现在项集里的电影，用它生成超集，更新该项集的计数。代码如下：

```
                for other_reviewed_movie in reviews - itemset:
                    current_superset = itemset | frozenset((other_
                                            reviewed_movie,))
                    counts[current_superset] += 1
```

函数最后检测达到支持度要求的项集，看它的频繁程度够不够，并返回其中的频繁项集。

```
    return dict([(itemset, frequency) for itemset, frequency in
counts.items() if frequency >= min_support])
```

创建循环，运行Apriori算法，存储算法运行过程中发现的新项集。循环体中，k 表示即将发现的频繁项集的长度，用键 k-1 可以从`frequent_itemsets`字典中获取刚发现的频繁项集。新发现的频繁项集以长度为键，将其保存到字典中。代码如下：

```
for k in range(2, 20):
    cur_frequent_itemsets =
      find_frequent_itemsets(favorable_reviews_by_users,
        frequent_itemsets[k-1],
          min_support)
      frequent_itemsets[k] = cur_frequent_itemsets
```

如果在上述循环中没能找到任何新的频繁项集，就跳出循环（输出信息，告知我们没能找到长度为 k 的频繁项集）。

```
    if len(cur_frequent_itemsets) == 0:
        print("Did not find any frequent itemsets of length {}".
          format(k))
        sys.stdout.flush()
        break
```

用`sys.stdout.flush()`方法，确保代码还在运行时，把缓冲区内容输出到终端。有时，在位于笔记本文件特定格子的大型循环中，代码结束运行前不会输出，用`flush`方法可以保证及时输出。但是，该方法不宜过多使用——`flush`操作（输出也是）所带

来的计算开销会拖慢程序的运行速度。

如果确实找到了频繁项集，我们也让程序输出信息，告知我们它会再次运行。因为算法运行时间很长，所以每隔一段时间输出一下状态是很有必要的！代码如下：

```
    else:
        print("I found {} frequent itemsets of length
        {}".format(len(cur_frequent_itemsets), k))
        sys.stdout.flush()
```

最后，循环结束，我们对只有一个元素的项集不再感兴趣——它们对生成关联规则没有用 处——生成关联规则至少需要两个项目。删除长度为1的项集。代码如下：

```
del frequent_itemsets[1]
```

上面这些代码要好几分钟才能运行完，老机器需要的时间更长。如果你在本地运行代码有问题，可以考虑使用云主机提升速度，具体细节请见附录。

上述代码返回了不同长度的1718个频繁项集。你也许会发现随着项集长度的增加，项集数随着可用规则的增加而增长一段时间后才开始变少，减少是因为项集达不到最低支持度要求。项集的减少是Apriori算法的优点之一。如果我们搜索所有可能的项集（不只是频繁项集的超集），判断多余项集的频繁程度需要成千上万次查询。

4.4　抽取关联规则

Apriori算法结束后，我们得到了一系列频繁项集，这还不算是真正意义上的关联规则，但是很接近了。频繁项集是一组达到最小支持度的项目，而关联规则由前提和结论组成。

我们可以从频繁项集中抽取出关联规则，把其中几部电影作为前提，另一部电影作为结论组成如下形式的规则：如果用户喜欢前提中的所有电影，那么他们也会喜欢结论中的电影。

每一个项集都可用这种方式生成一条规则。

下面的代码通过遍历不同长度的频繁项集，为每个项集生成规则。

```
candidate_rules = []
for itemset_length, itemset_counts in frequent_itemsets.items():
    for itemset in itemset_counts.keys():
```

然后，遍历项集中的每一部电影，把它作为结论。项集中的其他电影作为前提，用前提和结论组成备选规则。

```
        for conclusion in itemset:
            premise = itemset - set((conclusion,))
            candidate_rules.append((premise, conclusion))
```

这样就能得到大量备选规则。通过以下代码查看前五条规则。

```
print(candidate_rules[:5])
```

输出结果如下：

```
[(frozenset({79}), 258), (frozenset({258}), 79), (frozenset({50}),
  64), (frozenset({64}), 50), (frozenset({127}), 181)]
```

在上述这些规则中，第一部分（frozenset）是作为规则前提的电影编号，后面的数字表示作为结论的电影编号。第一组数据表示如果用户喜欢电影79，他很可能喜欢电影258。

接下来，计算每条规则的置信度，计算方法跟第1章类似，只不过要根据这里新的数据格式做些改动。

我们需要先创建两个字典，用来存储规则应验（正例）和规则不适用（反例）的次数。代码如下：

```
correct_counts = defaultdict(int)
incorrect_counts = defaultdict(int)
```

遍历所有用户及其喜欢的电影数据，在这个过程中遍历每条关联规则。

```
for user, reviews in favorable_reviews_by_users.items():
    for candidate_rule in candidate_rules:
        premise, conclusion = candidate_rule
```

```

```

测试每条规则的前提对用户是否适用。换句话说，用户是否喜欢前提中的所有电影。代码如下：

```
        if premise.issubset(reviews):
```

如果前提符合，看一下用户是否喜欢结论中的电影。如果是的话，规则适用，反之，规则不适用。

```
        if premise.issubset(reviews):
            if conclusion in reviews:
                correct_counts[candidate_rule] += 1
            else:
                incorrect_counts[candidate_rule] += 1
```

用规则应验的次数除以前提条件出现的总次数，计算每条规则的置信度。

```
rule_confidence = {candidate_rule: correct_counts[candidate_rule]
  / float(correct_counts[candidate_rule] +
    incorrect_counts[candidate_rule])
              for candidate_rule in candidate_rules}
```

对置信度字典进行排序后，输出置信度最高的前五条规则。

```
from operator import itemgetter
sorted_confidence = sorted(rule_confidence.items(),
  key=itemgetter(1), reverse=True)
for index in range(5):
    print("Rule #{0}".format(index + 1))
    (premise, conclusion) = sorted_confidence[index][0]
    print("Rule: If a person recommends {0} they will also
      recommend {1}".format(premise, conclusion))
    print(" - Confidence:
      {0:.3f}".format(rule_confidence[(premise, conclusion)]))
    print("")
```

结果如下：

```
Rule #1
Rule: If a person recommends frozenset({64, 56, 98, 50, 7}) they will
```

```
also recommend 174
 - Confidence: 1.000

Rule #2
Rule: If a person recommends frozenset({98, 100, 172, 79, 50, 56})
they will also recommend 7
 - Confidence: 1.000

Rule #3
Rule: If a person recommends frozenset({98, 172, 181, 174, 7}) they
will also recommend 50
 - Confidence: 1.000

Rule #4
Rule: If a person recommends frozenset({64, 98, 100, 7, 172, 50}) they
will also recommend 174
 - Confidence: 1.000

Rule #5
Rule: If a person recommends frozenset({64, 1, 7, 172, 79, 50}) they
will also recommend 181
 - Confidence: 1.000
```

输出结果中只显示电影编号，而没有显示电影名字，很不友好。我们下载的数据集中的u.items文件里存储了电影名称和编号（还有体裁等信息）。

用pandas从u.items文件加载电影名称信息。关于该文件和类别的更多信息请见数据集中的README文件。u.items文件为CSV格式，但是用竖线分隔数据。读取时需要指定分隔符，设置表头和编码格式。每一列的名称是从README文件中找到的。

```
movie_name_filename = os.path.join(data_folder, "u.item")
movie_name_data = pd.read_csv(movie_name_filename, delimiter="|",
  header=None, encoding = "mac-roman")

movie_name_data.columns = ["MovieID", "Title", "Release Date",
  "Video Release", "IMDB", "<UNK>", "Action", "Adventure",
    "Animation", "Children's", "Comedy", "Crime", "Documentary",
      "Drama", "Fantasy", "Film-Noir",
  "Horror", "Musical", "Mystery", "Romance", "Sci-Fi", "Thriller",
    "War", "Western"]
```

既然电影名称对于理解数据很重要，我们就来创建一个用电影编号获取名称的函数，以免去每次都要人工查找的烦恼。函数声明如下：

```
def get_movie_name(movie_id):
```

在数据框movie_name_data中查找电影编号，找到后，只返回电影名称列的数据。

```
    title_object = movie_name_data[movie_name_data["MovieID"] ==
      movie_id]["Title"]
```

我们用title_object的values属性获取电影名称（不是存储在title_object中的Series对象）。我们只对第一个值感兴趣——当然每个电影编号只对应一个名称！

```
    title = title_object.values[0]
```

函数最后返回电影名称。

```
    return title
```

调整之前的代码，这样就能在输出的规则中显示电影名称。请在IPython Notebook笔记本文件的新格子里输入以下代码。

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    (premise, conclusion) = sorted_confidence[index][0]
    premise_names = ", ".join(get_movie_name(idx) for idx
      in premise)
    conclusion_name = get_movie_name(conclusion)
    print("Rule: If a person recommends {0} they will
      also recommend {1}".format(premise_names, conclusion_name))
    print(" - Confidence: {0:.3f}".format(confidence[(premise,
      conclusion)]))
print("")
```

结果清楚多了（还有些小问题，暂时先忽略）。

```
Rule #1
Rule: If a person recommends Shawshank Redemption, The (1994), Pulp
Fiction (1994), Silence of the Lambs, The (1991), Star Wars (1977),
Twelve Monkeys (1995) they will also recommend Raiders of the Lost Ark
(1981)
 - Confidence: 1.000
```

```
Rule #2
Rule: If a person recommends Silence of the Lambs, The (1991), Fargo
(1996), Empire Strikes Back, The (1980), Fugitive, The (1993), Star
Wars (1977), Pulp Fiction (1994) they will also recommend Twelve
Monkeys (1995)
 - Confidence: 1.000

Rule #3
Rule: If a person recommends Silence of the Lambs, The (1991), Empire
Strikes Back, The (1980), Return of the Jedi (1983), Raiders of the
Lost Ark (1981), Twelve Monkeys (1995) they will also recommend Star
Wars (1977)
 - Confidence: 1.000

Rule #4
Rule: If a person recommends Shawshank Redemption, The (1994), Silence
of the Lambs, The (1991), Fargo (1996), Twelve Monkeys (1995), Empire
Strikes Back, The (1980), Star Wars (1977) they will also recommend
Raiders of the Lost Ark (1981)
 - Confidence: 1.000

Rule #5
Rule: If a person recommends Shawshank Redemption, The (1994), Toy
Story (1995), Twelve Monkeys (1995), Empire Strikes Back, The (1980),
Fugitive, The (1993), Star Wars (1977) they will also recommend Return
of the Jedi (1983)
 - Confidence: 1.000
```

评估

广义上讲，我们可以拿评估分类算法的那一套来用。训练前，留出一部分数据用于测试，评估发现的规则在测试集上的表现。

如果这样做的话，我们就需要计算每条规则在测试集中的置信度。

这里我们不打算用正式的评价指标，只是简单看一下每条规则的表现，寻找好的规则。

首先，抽取所有没有用于训练的数据作为测试集。训练集用前200名用户的打分数据，测试集就用剩下的数据。就训练集来说，我们会为该数据集中的每一位用户获取最喜欢的电影。代码如下：

```
test_dataset =
all_ratings[~all_ratings['UserID'].isin(range(200))]
test_favorable = test_dataset[test_dataset["Favorable"]]
test_favorable_by_users = dict((k, frozenset(v.values)) for k, v
  in test_favorable.groupby("UserID")["MovieID"])
```

接着，计算规则应验的数量，方法跟之前相同。唯一的不同就是这次使用的是测试数据而不是训练数据。代码如下：

```
correct_counts = defaultdict(int)
incorrect_counts = defaultdict(int)
for user, reviews in test_favorable_by_users.items():
    for candidate_rule in candidate_rules:
        premise, conclusion = candidate_rule
        if premise.issubset(reviews):
            if conclusion in reviews:
                correct_counts[candidate_rule] += 1
            else:
                incorrect_counts[candidate_rule] += 1
```

接下来，计算所有应验规则的置信度。

```
test_confidence = {candidate_rule: correct_counts[candidate_rule]
  / float(correct_counts[candidate_rule] + incorrect_counts
    [candidate_rule])
  for candidate_rule in rule_confidence}
```

最后，输出用电影名称而不是电影编号表示的最佳关联规则。

```
for index in range(5):
    print("Rule #{0}".format(index + 1))
    (premise, conclusion) = sorted_confidence[index][0]
    premise_names = ", ".join(get_movie_name(idx) for idx in
      premise)
    conclusion_name = get_movie_name(conclusion)
    print("Rule: If a person recommends {0} they will also
      recommend {1}".format(premise_names, conclusion_name))
    print(" - Train Confidence:
      {0:.3f}".format(rule_confidence.get((premise, conclusion),
        -1)))
    print(" - Test Confidence:
     {0:.3f}".format(test_confidence.get((premise, conclusion),
        -1)))
     print("")
```

我们来看一下在新数据集上哪些规则最适用。

```
Rule #1
Rule: If a person recommends Shawshank Redemption, The (1994), Pulp
```

```
Fiction (1994), Silence of the Lambs, The (1991), Star Wars (1977),
Twelve Monkeys (1995) they will also recommend Raiders of the Lost Ark
(1981)
 - Train Confidence: 1.000
 - Test Confidence: 0.909

Rule #2
Rule: If a person recommends Silence of the Lambs, The (1991), Fargo
(1996), Empire Strikes Back, The (1980), Fugitive, The (1993), Star
Wars (1977), Pulp Fiction (1994) they will also recommend Twelve
Monkeys (1995)
 - Train Confidence: 1.000
 - Test Confidence: 0.609

Rule #3
Rule: If a person recommends Silence of the Lambs, The (1991), Empire
Strikes Back, The (1980), Return of the Jedi (1983), Raiders of the
Lost Ark (1981), Twelve Monkeys (1995) they will also recommend Star
Wars (1977)
 - Train Confidence: 1.000
 - Test Confidence: 0.946

Rule #4
Rule: If a person recommends Shawshank Redemption, The (1994), Silence
of the Lambs, The (1991), Fargo (1996), Twelve Monkeys (1995), Empire
Strikes Back, The (1980), Star Wars (1977) they will also recommend
Raiders of the Lost Ark (1981)
 - Train Confidence: 1.000
 - Test Confidence: 0.971

Rule #5
Rule: If a person recommends Shawshank Redemption, The (1994), Toy
Story (1995), Twelve Monkeys (1995), Empire Strikes Back, The (1980),
Fugitive, The (1993), Star Wars (1977) they will also recommend Return
of the Jedi (1983)
 - Train Confidence: 1.000
 - Test Confidence: 0.900
```

举例来说，第二条规则，在训练集中置信度为1，但在测试集上正确率只有60%。但前十条规则中，其他几条规则在测试集上置信度也很高，用它们来推荐电影效果不错。

> 浏览剩余规则，你可能发现有些规则的置信度为-1。置信度一般为0到1之间的值。负数表示这条规则在测试集中根本不存在。

4.5 小结

本章把亲和性分析用到电影推荐上，从大量电影打分数据中找到可用于电影推荐的关联规则。整个过程分为两大阶段。首先，借助Apriori算法寻找数据中的频繁项集。然后，根据找到的频繁项集，生成关联规则。

由于数据集较大，Apriori算法就很有必要。虽然第1章使用最笨的方法来寻找规则，但是在这里就不适用了，由于时间复杂度呈指数级增长，我们需要寻找更巧妙的解决方案。这种现象在数据挖掘中很常见：虽然可以用最笨的方法穷尽所有情况来解决问题，但在数据量很大的情况下必须要使用更灵活、更快捷的算法。

我们用一部分数据作为训练集以发现关联规则，在剩余数据——测试集上进行测试。可以用前几章的交叉检验方法对每条规则的效果进行更为充分的评估。

到目前为止，我们使用的数据集都有很明显的特征。然而，不是所有数据集都是这个样子。下一章将学习用`scikit-learn`的转换器（第3章曾介绍过）从数据中抽取特征。还将讨论怎样实现自己的转换器和扩展已有转换器，并介绍相关概念。

第 5 章　用转换器抽取特征

前几章所用到的数据集都是用特征来描述的。虽然上一章的数据集为交易类型的数据，形式稍有不同，但归根结底还是一种用特征来描述的数据集。

除此之外，还有很多其他类型的数据集，比如文本、图像、声音、视频甚至是真实的物体。然而，大多数数据挖掘算法都依赖于数值或类别型特征。这表明在使用数据挖掘算法处理它们之前，需要找到一种表示它们的方法。

本章所讨论的是如何从数据集中抽取数值和类别型特征，并选出最佳特征，前提是数据集确实包含这些特征。我们还会介绍特征抽取的常用模式和技巧。

本章主要介绍以下几个概念。

- 从数据集中抽取特征
- 创建新特征
- 选取好特征
- 创建转换器，处理数据集

5.1　特征抽取

特征抽取是数据挖掘任务最为重要的一个环节，一般而言，它对最终结果的影响要高过数据挖掘算法本身。不幸的是，关于怎样选取好的特征，还没有严格、快捷的规则可循，其实这也是数据挖掘科学更像是一门艺术的所在。创建好的规则离不开直觉，还需要专业领域知识和数据挖掘经验，光有这些还不够，还得不停地尝试、摸索，在试错中前进，有时多少还要靠点运气。

5.1.1　在模型中表示事实

不是所有的数据集都是用特征来表示的。数据集可以是一位作家所写的全部书籍，也可以是1979年以来所有电影的胶片，还可以是馆藏的

一批历史文物。

我们可能想对以上数据集做数据挖掘。对于作家的作品，我们想知道这位作家都写过哪些主题；对于电影，我们想知道女性形象是怎么塑造的；对于文物，我们想知道它们是何方产物。我们没办法把实物直接塞进决策树来得到结果。

只有先把现实用特征表示出来，才能借助数据挖掘的力量找到问题的答案。特征可用于建模，模型以机器挖掘算法能够理解的近似的方式来表示现实。因此可以说，模型描述了客观世界中对象的某些方面，是它的一个简化版本。举例来说，象棋就是对过往战事简化后得到的一种模型。

特征选择的另一个优点在于：降低真实世界的复杂度，模型比现实更容易操纵。想象一下，向一个没有任何背景知识的人介绍一种新事物，要给出恰如其分、精确又不失全面的描述需要提供多少信息。我们可能需要介绍其尺寸、重量、质地、成分、年龄、瑕疵、用途、产地等信息。

实物的复杂性对目前的算法而言过于复杂，我们退而求其次，使用更为简洁的模型来表示实物。

简化要以数据挖掘应用的目标为核心。本书后面会讲到聚类，对这类应用而言，特征选取至关重要，如果随意选取特征，分簇结果因选择的特征而异，呈现出很强的随机性。

降低复杂性有好处，但也有不足，简化会忽略很多细节，甚至会抛弃很多对数据挖掘算法能起到帮助作用的信息。

我们应该始终关注怎么用模型来表示现实，要多考虑数据挖掘的目标，而不是轻率地用我们过去用过的特征。要经常想想我们的目标是什么。第3章创建特征时充分考虑目标（预测赢家），使用篮球比赛领域相关知识，找出哪些因素与比赛结果密切相关，据此再创建新特征。

> 不是所有的特征必须是数值或类别型值，直接用于文本、图像和其他数据结构的算法已经研究出来了。不幸的是，篇幅有限，本书主要讲解数值和类别型特征，不涉及上述算法。

用Adult数据集讲解如何借助特征为复杂的现实世界建模再好不过。

我们这里数据挖掘的目标是，预测一个人是否年收入多于五万美元。数据集下载地址为http://archive.ics.uci.edu/ml/datasets/Adult，点击Data Folder链接。下载adult.data和adult.names文件，在数据集文件夹（主目录下Data文件夹）中创建Adult文件夹，将这两个文件放到里面。

数据集用特征描述了一个个活生生的人及其他们所处的环境、背景和生活状况。

打开IPython Notebook，新建一个笔记本文件用于本章实验，导入pandas模块，指定数据集文件的路径，用pandas加载数据集文件。

```
import os
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "Adult")
adult_filename = os.path.join(data_folder, "adult.data")
```

像以前一样使用pandas，用`read_csv`加载文件，如下所示。

```
adult = pd.read_csv(adult_filename, header=None,
    names=["Age", "Work-Class", "fnlwgt",
    "Education", "Education-Num",
    "Marital-Status", "Occupation",
    "Relationship", "Race", "Sex",
    "Capital-gain", "Capital-loss",
    "Hours-per-week", "Native-Country",
    "Earnings-Raw"])
```

大部分代码在上一章都见过。

adult.data文件末尾有两处空行。pandas默认把倒数第二行空行作为一条有效数据读入（只不过各列的值为空）。我们需要删除包含无效数字的行（设置`inplace`参数为真，表示改动当前数据框，而不是新建一个）。

```
adult.dropna(how='all', inplace=True)
```

输入下面代码并运行，查看所有的特征名称。

```
adult.columns
```

下面为所有保存在pandas的`Index`对象中的特征名称。

```
Index(['Age', 'Work-Class', 'fnlwgt', 'Education',
'Education-Num', 'Marital-Status', 'Occupation', 'Relationship',
'Race', 'Sex', 'Capital-gain', 'Capital-loss', 'Hours-per-week',
'Native-Country', 'Earnings-Raw'], dtype='object')
```

5.1.2 通用的特征创建模式

纵然有数不清的特征创建方法，但其中有一些是适用于不同学科的通用模式。然而，选择合适的特征是项技术活，需要考虑特征和最终结果之间的相互关系。正如谚语所说的，不要用封面来评价书的好坏——如果你对书的内容感兴趣，书的大小可能不用考虑。

一些通用特征关注的是所研究对象的物理属性。

- 空间属性，比如对象的长、宽、高
- 重量和（或）密度
- 年龄、使用年限或成分
- 种类
- 品质

另一些特征可能与对象的使用或历史相关。

- 生产商、出版商或创造者
- 生产时间
- 使用方法

还有一些特征从组成成分角度描述对象。

- 次级成分的频率，如一本书中某个单词的词频
- 次级成分的数量和（或）次级成分种类的数量
- 次级成分的平均大小，例如平均句长

序数特征可对其排序、分组。正如前几章所见到的，特征可以是数值或类别型特征。数值特征通常被看作是有顺序的（ordinal）。例如，爱丽丝、鲍勃、查理三个人身高分别为1.5m、1.6m和1.7m。我们可以说爱丽丝和鲍勃，比起爱丽丝和查理，在身高上更相似。

上一节我们导入的数据集，包含连续特征、序数特征。例如，`Hours-per-week`特征表示一个人每周的工作时间。这个特征可用来计算平均工作时间、标准差、最大值和最小值。pandas提供了常见统计量的计算方法。

```
adult["Hours-per-week"].describe()
```

输出结果让我们对这个特征有了更多了解。

```
count    32561.000000
mean        40.437456
std         12.347429
min          1.000000
25%         40.000000
50%         40.000000
75%         45.000000
max         99.000000
dtype: float64
```

这些统计方法对其他特征可能没有意义。例如，计算受教育程度的和就讲不通。

还有些特征，它们虽然不是数值，但仍然属于序数值，比如Adult数据集中的`Education`（教育）特征，学士学位比高中毕业，在受教育程度上，要高一个层次，而高中毕业又比没有完成高中学业高一个层次。计算它俩的均值就没有意义，但是我们可以找到一种近似的表示方法。数据集有个叫作`Education-Num`（受教育年限）的特征，它的取值基本上就是每个教育阶段的年限。这样我们就能快速计算受教育年限的中位数。

```
adult["Education-Num"].median()
```

结果为10，或者可以表述为刚好读完高一。如果没有受教育年限数据，我们为不同教育阶段指定数字编号，也可以计算中位数。

特征也可以是类别型的。例如，一个球可以是网球、棒球、足球或其他种类。类别型特征也可以称为名义特征（nominal）。对于名义特征而言，几个值之间要么相同要么不同。球类可以根据大小或重量排序，但是无法仅根据类别值进行量上的比较。网球不是棒球，也不是足球。我们可以说比起足球，网球的大小跟棒球更接近，但是无法在类别上做类似的区分——它们要么是同一类，要么不是。

类别型特征二值化后就变成了数值型特征，第3章中曾经用过。对于上述球的种类，可以创建三个二值特征："是网球""是棒球"和"是足球"。如果是网球的话，特征向量就是[1, 0, 0]，棒球[0, 1, 0]，足球[0, 0, 1]。这些特征都只有两个取值，很多算法都可以把它们作为连续型特征使用。二值化的好处是，便于直接进行数字上的比较（例如计算个体之间的距离）。

Adult数据集包含一些类别型特征，例如`Work-Class`（工作），其中它的有些值可以进行量级上的比较，比如不同的就业状况影响收入（例如有工作的人比没有工作的人收入高）。再比如，州政府职员的工资不太可能比私企职工工资高。

用数据框的`unique`函数就能得到所有的工作情况。

```
adult["Work-Class"].unique()
```

输出结果如下：

```
array([' State-gov', ' Self-emp-not-inc', ' Private', ' Federal-gov',
       ' Local-gov', ' ?', ' Self-emp-inc', ' Without-pay',
       ' Never-worked', nan], dtype=object)
```

数据集部分数据缺失，但不会影响这里的计算。

同理，数值型特征也可以通过离散化过程转换为类别型特征，第4章中曾用过。例如，高于1.7m的人，我们称其为高个子，低于1.7m的叫作矮个子。这样就得到了类别型特征（虽然是序数值类型）。在这个过程中确实丢失了一些信息。例如，有两个人，身高分别为1.69m和1.71m，这两个个子差不多的将被分到两个截然不同的类别里。而身高仅为1.2m的将会被认为与身高1.69m的"差不多高"！细节的丢失是离散化不好的一面，也是建模时需要考虑解决的问题。

对于Adult数据集，我们可以创建`LongHours`（长时）特征，用它来

表示一个人每周工作时长是否多于40小时。这样就把连续值（Hours-per-week，每周工作时长）转换为类别型特征。

```
adult["LongHours"] = adult["Hours-per-week"] > 40
```

5.1.3　创建好的特征

建模过程中，需要对真实世界中的对象进行简化，这会导致信息的丢失，这也就是为什么没有一套能够用于任何数据集的通用的数据挖掘方法。数据挖掘的行家里手需要拥有数据来源领域的知识，没有的话，要积极去掌握。他们弄清楚问题是什么，了解有哪些可用数据后，在此基础上，才能创建解决问题所需的模型。

例如，身高这个特征描述的是一个人外表的某个方面，但是不能说明这个人学习成绩如何，所以预测学习成绩时，我们没必要去测量每个人的身高。

这正是数据挖掘为什么变得更像是一门艺术而不是科学的原因所在。抽取好的特征难度不小，该课题具有重要研究意义，因此它获得了研究人员的持续关注。选择好的分类算法也可以提升数据挖掘应用的效果，但最好还是通过选用好的特征来达到同样的目的。

不论是设计什么数据挖掘应用，首先都应该确定大致的方向，然后再设计方法实现目标。知道自己的目标之后，就能确定所需要的特征类型和算法，对最终结果也做到心中有数。

5.2　特征选择

通常特征数量很多，但我们只想选用其中一小部分。有如下几个原因。

- 降低复杂度：随着特征数量的增加，很多数据挖掘算法需要更多的时间和资源。减少特征数量，是提高算法运行速度，减少资源使用的好方法。

- 降低噪音：增加额外特征并不总会提升算法的表现。额外特征可能扰乱算法的正常工作，这些额外特征间的相关性和模式没有实际应用价值（这种情况在小数据集上很常见）。只选择合适的特征有助于减少出现没有实际意义的相关性的几率。

- 增加模型可读性：根据成千上万个特征创建的模型来解答一个问题，对计算机来说很容易，但模型对我们自己来说就晦涩无比。因此，使用更少的特征，创建我们自己可以理解的模型，就很有必要。

有些分类算法确实很强壮，能够处理噪音问题，特征再多也不在话下，但是选用干净的数据，选取更具描述性的特征，对算法效果提升很有帮助。

在开展数据挖掘工作前，有些基础性测试我们要做，比如确保特征值是不同的。如果特征值都相同，就跟没提供什么信息一样，挖掘就失去了意义。

scikit-learn中的VarianceThreshold转换器可用来删除特征值的方差达不到最低标准的特征。下面通过代码来讲下它的用法，首先用numpy创建一个简单的矩阵。

```
import numpy as np
X = np.arange(30).reshape((10, 3))
```

上述矩阵包含0到29，共30个数字，分为3列10行。可以把它看成一个有10个个体、3个特征的数据集。

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23],
       [24, 25, 26],
       [27, 28, 29]])
```

接着，把所有第二列的数值都改为1。

```
X[:,1] = 1
```

第一、三列特征值方差很大，而第二列方差为0。

```
array([[ 0, 1,  2],
```

```
       [ 3,  1,   5],
       [ 6,  1,   8],
       [ 9,  1,  11],
       [12,  1,  14],
       [15,  1,  17],
       [18,  1,  20],
       [21,  1,  23],
       [24,  1,  26],
       [27,  1,  29]])
```

这时再来创建VarianceThreshold转换器，用它处理数据集。

```
from sklearn.feature_selection import VarianceThreshold
vt = VarianceThreshold()
Xt = vt.fit_transform(X)
```

输出Xt后，我们发现第二列消失了。

```
array([[ 0,   2],
       [ 3,   5],
       [ 6,   8],
       [ 9,  11],
       [12,  14],
       [15,  17],
       [18,  20],
       [21,  23],
       [24,  26],
       [27,  29]])
```

输出每一列的方差。

```
print(vt.variances_)
```

下面输出结果表明第一、三列包含有价值信息，第二列方差为0，不包含具有区别意义的信息。

```
array([ 74.25,  0.  ,  74.25])
```

无论什么时候，拿到数据后，先做下类似简单、直接的分析，对数据集的特点做到心中有数。方差为0的特征不但对数据挖掘没有丝毫用处，相反还会拖慢算法的运行速度。

选择最佳特征

特征很多的情况下，怎么选出最佳的几个，可有点难度。它与解决数据挖掘问题自身相关，计算量很大。正如第4章讲到的，随着特征数量的增加，寻找子集的任务复杂度呈指数级增长。寻找最佳特征组合的时间复杂度同样是指数级增长的。

其中一个变通方法是不要找表现好的子集，而只是去找表现好的单个特征（单变量），依据是它们各自所能达到的精确度。分类任务通常是这么做的，我们一般只要测量变量和目标类别之间的某种相关性就行。

scikit-learn提供了几个用于选择单变量特征的转换器，其中SelectKBest返回*k*个最佳特征，SelectPercentile返回表现最佳的前r%个特征。这两个转换器都提供计算特征表现的一系列方法。

单个特征和某一类别之间相关性的计算方法有很多。最常用的有卡方检验（χ^2）。其他方法还有互信息和信息熵。

我们可以测试单个特征在Adult数据集上的表现。首先，选取下述特征，从pandas数据框中抽取一部分数据。

```
X = adult[["Age", "Education-Num", "Capital-gain", "Capital-loss",
  "Hours-per-week"]].values
```

接着，判断Earnings-Raw（税前收入）是否达到五万美元，创建目标类别列表。如果达到，类别为True，否则，类别为False。代码如下：

```
y = (adult["Earnings-Raw"] == ' >50K').values
```

再使用SelectKBest转换器类，用卡方函数打分，初始化转换器。

```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
transformer = SelectKBest(score_func=chi2, k=3)
```

调用fit_transform方法，对相同的数据集进行预处理和转换。结果为分类效果较好的三个特征。代码如下：

```
Xt_chi2 = transformer.fit_transform(X, y)
```

生成的矩阵只包含三个特征。我们还可以得到每一列的相关性，这样就可以知道都使用了哪些特征。还是看下代码：

```
print(transformer.scores_)
```

输出结果如下：

```
[   8.60061182e+03    2.40142178e+03    8.21924671e+07    1.37214589e+06
    6.47640900e+03]
```

相关性最好的分别是第一、三、四列，分别对应着Age（年龄）、Capital-Gain（资本收益）和Capital-Loss（资本损失）三个特征。从单变量特征选取角度来说，这些就是最佳特征。

如果你想了解更多关于Adult数据集各特征的相关信息，请查看adult.names文件，里面有更多介绍及相关文献。

还可以用其他方法计算相关性，比如皮尔逊（Pearson）1相关系数。用于科学计算的SciPy库（scikit-learn在它基础上开发）实现了该方法。

1卡尔·皮尔逊（Karl Pearson，1857—1936），原名Carl，英国数学家、生物统计学家。——译者注

如果你的计算机可以运行scikitlearn，那么就可以运行SciPy。运行下面这个例子不需要装额外的库。

首先，从SciPy导入pearsonr函数。

```
from scipy.stats import pearsonr
```

pearsonr函数的接口几乎与scikit-learn单变量转换器接口一致，该函数接收两个数组（当前例子中为x和y）作为参数，返回两个数组：每个特征的皮尔逊相关系数和p值（p-value）。前面用到的

chi2函数使用的接口符合要求，因此我们直接把它传入到SelectKBest函数中。

SciPy的pearsonr函数参数为两个数组，但要注意的是第一个参数x为一维数组。我们来实现一个包装器函数，这样就能像前面那样处理多维数组。函数声明如下：

```
def multivariate_pearsonr(X, y):
```

创建scores和pvalues数组，遍历数据集的每一列。

```
    scores, pvalues = [], []
    for column in range(X.shape[1]):
```

只计算该列的皮尔逊相关系数和p值，并将其存储到相应数组中。

```
        cur_score, cur_p = pearsonr(X[:,column], y)
        scores.append(abs(cur_score))
        pvalues.append(cur_p)
```

> 皮尔逊相关系数为-1到1之间的任意值。值为1，表示两个变量之间绝对正相关，值为-1，绝对负相关，即一个变量值越大，另一个变量值就越小，反之亦然。这样的特征确实能反应两个变量之间的关系，但是根据大小进行排序，这些值因为是负数而排在后面，可能会被舍弃不用。因此，我们对皮尔逊相关系数取绝对值后，再保存到scores数组中，而不是使用原始值。

函数最后返回包含皮尔逊相关系数和p值的元组。

```
    return (np.array(scores), np.array(pvalues))
```

现在，我们就可以像之前那样使用转换器类，根据皮尔逊相关系数对特征进行排序。

```
transformer = SelectKBest(score_func=multivariate_pearsonr, k=3)
Xt_pearson = transformer.fit_transform(X, y)
print(transformer.scores_)
```

```
```

返回的特征跟用卡方检验计算相关性得到的特征不一样！这回得到的是第一、二、五列：Age、Education和Hours-per-week。这表明哪些特征是最好的这个问题没有标准答案——取决于度量标准。

我们在分类器中看看哪个特征集合效果更好。请注意实验结果也只表明对于特定分类器和（或）特征子集效果更好——在数据挖掘领域，一种方法在任何情况下都比另一种方法好的情况几乎没有！实验代码如下：

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.cross_validation import cross_val_score
clf = DecisionTreeClassifier(random_state=14)
scores_chi2 = cross_val_score(clf, Xt_chi2, y, scoring='accuracy')
scores_pearson = cross_val_score(clf, Xt_pearson, y,
  scoing='accuracy')
```

chi2方法的平均正确率为0.83，而皮尔逊相关系数正确率为0.77。用卡方检验得到的特征组合效果更好！

别忘了本章数据挖掘任务目标是预测收入。使用特征选择技术，找到好的特征组合，只用三个特征，就能达到83%的正确率！

5.3　创建特征

有时，仅仅选择已有特征不够用。这时，我们就可以用不同的方法从已有特征中发掘新特征，比如前面用到的一位有效编码（one-hot encoding），我们可以用它把有A、B、C三个选项的类别型特征转换为三个新的特征："是A吗？""是B吗？"和"是C吗？"。

创建新特征看上去没必要，也没什么好处——毕竟，这些信息原本就在数据集里，我们只是要用而已。然而，特征之间相关性很强，或者特征冗余，会增加算法处理难度。

出于这个原因，创建特征就很有必要，很多方法也应运而生。

接下来，我们要加载一个新数据集，正好也该创建一个新的IPython Notebook笔记本文件了。从http://archive.ics.uci.edu/ml/datasets/Internet+Advertisements下载Advertisements（广告）数据集，保存到自己主目录下的Data文件夹中。

接着，用pandas加载数据集。我们还是先指定文件的路径。

```
import os
import numpy as np
import pandas as pd
data_folder = os.path.join(os.path.expanduser("~"), "Data")
data_filename = os.path.join(data_folder, "Ads", "ad.data")
```

数据集存在几个问题，加载过程需要我们做些处理。问题一，前几个特征是数值，但是pandas会把它们当成字符串。要修复这个问题，我们需要编写将字符串转换为数字的函数，该函数能够把只包含数字的字符串转换为数字，把其余的转化为“NaN”2（“Not a Number”，不是一个数字），表示参数值无法转换为数字，该值类似于其他编程语言中的none或null。

> 2JavaScript语言中也有该类型，详见《JavaScript高级程序设计（第3版）》 第29页。——译者注

问题二，数据集中有些值缺失，缺失的值用“?”表示。幸运的是，问号不会被转换为浮点型数据，因此，我们也可以把它们转换为“NaN”。后续章节会讲到其他处理缺失值的方法。

转换函数声明如下：

```
def convert_number(x):
```

先试着把字符串转换为数字，如果失败就要处理异常，所以我们这里使用`try/except`结构，捕获`ValueError`异常（字符串无法转换为数字时，抛出异常）。

```
    try:
        return float(x)
    except ValueError:
```

在函数的最后，如果转换失败，返回`numpy`库中的“NaN”类型。

```
        return np.nan
```

我们来创建一个字典存储所有特征及其转换结果，我们想把所有的特

征值转换为浮点型。

```
    converters = defaultdict(convert_number)
```

我们还想把最后一列（编号为#1558）的值转化为0或1，该列表示每条数据的类别。在Adult数据集中，我们专门创建了一列表示类别。对于当前实验，导入数据时，顺便把类别这一列各个类别值由字符串转换为数值。

```
converters[1558] = lambda x: 1 if x.strip() == "ad." else 0
```

可以用`read_csv`加载数据集了，在参数中指定我们刚创建的转换函数。

```
ads = pd.read_csv(data_filename, header=None, converters=converters)
```

数据集很大，有1559列，2000多条数据。先来看下前五条数据，在笔记本的新格子中输入并运行`ads[:5]`。

	0	1	2	3	4	5	6	7	8	9	...	1549	1550	1551	1552	1553	1554	1555	1556	1557	1558
0	125	125	1.0000	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
1	57	468	8.2105	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
2	33	230	6.9696	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
3	60	468	7.8000	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
4	60	468	7.8000	1	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1

数据集所描述的是网上的图像，目标是确定图像是不是广告。

从数据集表头中无法获知每列数据的含义。你下载的另外两个文件ad.DOCUMENTATION和ad.names有更多信息。前三 个特征分别指图像的高度、宽度和宽高比。最后一列是数据的类别，1表示是广告，0表示不是广告。

其余特征取值为1的话，表示图像的URL、`alt`属性值或图像标题中包含某个特定的可以帮助判断是不是广告的词语，比如像sponsor（赞助商）。很多特征重合度很高，因为它们组合在一起恰好是其他特征。因此，数据集包含大量冗余信息。

将数据集加载到pandas之后，我们再来抽取用于分类算法的`x`矩阵和`y`数组，`x`矩阵为数据框除去最后一列的所有列，`y`数组包含数据框的最后一列，也就是列编号为#1558的列。代码如下。

```
X = ads.drop(1558, axis=1).values
y = ads[1558]
```

主成分分析

有些数据集，特征之间联系紧密。例如，只有1个档的微型单座赛车，速度和油耗两者关系就极为密切。特征间的相关性信息在某些场合会很有用，但是数据挖掘算法通常不需要这些冗余信息。

Advertisements数据集中有些特征相关性特别大，因为很多关键词在图像的`alt`属性值及图像的标题中都会使用。

主成分分析算法（Principal Component Analysis，PCA）的目的是找到能用较少信息描述数据集的特征组合。它意在发现彼此之间没有相关性、能够描述数据集的特征，确切说这些特征的方差跟整体方差没有多大差距，这样的特征也被称为主成分。这也就意味着，借助这种方法，就能通过更少的特征捕获到数据集的大部分信息。

PCA跟其他转换器用法类似。它只有主成分数量这一个参数。它默认会返回数据集中的所有特征。然而，PCA会对返回结果根据方差大小进行排序，返回的第一个特征方差最大，第二个特征方差稍小，以此类推。因此，前几个特征往往就能够解释数据集的大部分信息。请见代码：

```
from sklearn.decomposition import PCA
pca = PCA(n_components=5)
Xd = pca.fit_transform(X)
```

返回的结果`Xd`矩阵只有五个特征，但是不容小觑，我们看一下每个特征的方差。

```
np.set_printoptions(precision=3, suppress=True)
pca.explained_variance_ratio_
```

结果`array([ 0.854, 0.145, 0.001, 0. , 0. ])`表明第一个特征的方差对数据集总体方差的贡献率为85.4%。第二个为14.5%。第四个特征方差贡献率不可能高于1‰，后面1553个特征则更少。

用PCA算法处理数据一个不好的地方在于，得到的主成分往往是其他几个特征的复杂组合，例如，上述第一个特征就是通过为原始数据集的1558个特征（虽然很多特征值为0）分别乘以不同权重得到的，前三个特征的权重依次为-0.092、-0.995和-0.024。经过某种组合得到的特征，如果没有丰富的研究经验，理解起来很困难。

用PCA算法得到的数据创建模型，不仅能够近似地表示原始数据集，还能提升分类任务的正确率。

```
clf = DecisionTreeClassifier(random_state=14)
scores_reduced = cross_val_score(clf, Xd, y, scoring='accuracy')
```

正确率为0.9356，比起只用所有的原始特征效果（稍）好。PCA算法不是说都能带来效果上的提升，但多数情况是有好处的。

> 我们这里用PCA算法来减少数据集中的冗余特征。一般来说，你不能用它来解决数据挖掘实验的过拟合问题。原因是PCA没有将类别考虑进去。更佳解决方案是正则化，简介及代码请见 http://blog.datadive.net/selectinggood-features-part-ii-linear-models-and-regularization/。

PCA算法的另一个优点是，你可以把抽象难懂的数据集绘制成图形。例如，把PCA返回的前两个特征做成图形。

首先，告诉IPython在当前笔记本作图，导入`pyplot`。

```
%matplotlib inline
from matplotlib import pyplot as plt
```

接着，获取数据集中类别的所有取值（只有两个：是广告和不是广告）。

```
classes = set(y)
```

指定在图形中用什么颜色表示这两个类别。

```
colors = ['red', 'green']
```

用zip函数将这两个列表组合起来，同时遍历。

```
for cur_class, color in zip(classes, colors):
```

为属于当前类别的所有个体创建遮罩层。

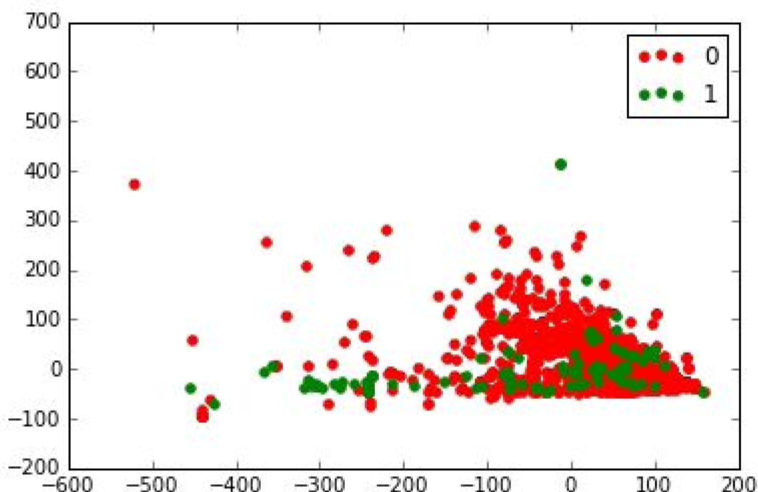
```
    mask = (y == cur_class).values
```

使用pyplot的scatter函数显示它们的位置。图中的x和y的值为前两个特征。

```
    plt.scatter(Xd[mask,0], Xd[mask,1], marker='o', color=color,
      label=int(cur_class))
```

最后，在循环的外面，创建图示，显示图像，从中就能看到分属两个类别的个体。

```
plt.legend()
plt.show()
```



5.4　创建自己的转换器

随着数据集复杂程度的提高和类型的变化，你可能发现，现成的特征抽取转换器不能满足需求了。比如，第7章从图这种数据结构里抽取特征，就需要自己编写转换器。

转换器像极了转换函数。它接收一种形式的数据，输出另外一种形式。转换器可以用训练集训练，训练得到的参数可以用来转换测试数据集。

转换器的API（应用编程接口）很简单。它接收一种特定格式的数据，输出一种格式的数据（可能与输入数据的格式相同，也可能不同）。别的就没有什么特殊要求了。

5.4.1　转换器API

转换器有两个关键函数。

- `fit()`：接收训练数据，设置内部参数。
- `transform()`：转换过程。接收训练数据集或相同格式的新数据集。

`fit()`和`transform()`函数的输入应该为同一种数据类型，但是`transform()`可以返回不同的数据类型。

我们现在就通过实现一个简单的转换器，来说明下API的使用。转换器接收`numpy`数组作为输入，根据均值将其离散化。任何高于均值的特征值（训练集中）替换为1，小于或等于均值的替换为0。

我们曾经用pandas对Adult数据集做过类似的转换：用`Hours-per-week`特征创建了`LongHours`特征，后者表示每周工作时长是否在40小时以上。即将编写的转换器有两个不同点。第一，接口与`scikit-learn`接口一致，便于在流水线中使用。第二，使用均值而不是固定的值（在`LongHours`例子中使用40作为将特征二值化的阈值）。

5.4.2　实现细节

打开分析Adult数据集时用到的笔记本文件。点击Cell 菜单，选择Run All，运行所有单元格。

首先，导入用来设置API 的`TransformerMixin`类。Python语言（相对于Java之类的语言）没有严格的接口规范，使用类似`mixin`这

样的类，scikit-learn就会把我们封装的类当作是转换器。我们还需要导入用来判断输入类型是否合法的函数，稍后就会用到。

导入所需模块。

```
from sklearn.base import TransformerMixin
from sklearn.utils import as_float_array
```

接着，创建继承自mixin的类。

```
class MeanDiscrete(TransformerMixin):
```

我们需要定义接口规范与API一致的fit和transform函数。在fit函数中，计算出数据集的均值，用内部变量保存该值。函数声明如下：

```
    def fit(self, X):
```

fit函数中，要确保X数据集可以处理，我们用到了as_float_array函数（尝试对X进行转换，例如X是浮点数列表时就可以对其进行转换）。

```
        X = as_float_array(X)
```

计算数组的均值，用内部变量保存该值。如果X是多变量数组，数组self.mean保存的是每个特征的均值。

```
        self.mean = X.mean(axis=0)
```

fit函数需要返回它本身，确保在转换器中能够进行链式调用（例如调用ransformer. fit(X).transform(X)）。函数返回语句如下：

```
        return self
```

接着，新建transform函数，参数类型与fit函数相同，检查下输入

是否合法。

```
    def transform(self, X):
         X = as_float_array(X)
```

输入必须是numpy数组（或等价的数据结构），除此之外，还需要检查输入的数据列数是否一致。X中的特征数应该与训练集中的特征数一致。

```
        assert X.shape[1] == self.mean.shape[0]
```

函数最后返回X中大于均值的数据。

```
        return X > self.mean
```

初始化一个实例，用它来对X数组进行转换。

```
mean_discrete = MeanDiscrete()
X_mean = mean_discrete.fit_transform(X)
```

5.4.3　单元测试

创建完函数和类后，最好是做下单元测试。单元测试，顾名思义，就是对代码中完成一个功能的代码单元进行测试。在这里，测试下转换器的功能是否符合预期。

充分的测试应当在独立于现有代码的前提下进行。为保证测试的有效性，最好使用另一种编程语言或方法来进行相同的计算。我们这里使用Excel创建数据集，再计算每一列的均值，接下来跟程序计算结果相比较。

单元测试应该短小、运行速度快。因此，我们选用少量数据进行测试。本例用于测试的数据集保存在前面创建的Xt变量中，在测试中会重新创建这个数据集。该数据集两个特征的均值分别为13.5和15.5。

为了创建单元测试，我们从numpy的testing模块中导入assert_array_equal函数，它用来检测两个数组是否相等。

```

```

```
from numpy.testing import assert_array_equal
```

接下来，创建测试函数，注意文件名以“test_”开头，这种命名方法便于工具自动查找并运行测试。代码如下：

```
def test_meandiscrete():
    X_test = np.array([[ 0,  2],
                       [ 3,  5],
                       [ 6,  8],
                       [ 9, 11],
                       [12, 14],
                       [15, 17],
                       [18, 20],
                       [21, 23],
                       [24, 26],
                       [27, 29]])
```

创建转换函数实例，用测试数据进行训练。

```
    mean_discrete = MeanDiscrete()
    mean_discrete.fit(X_test)
```

通过与在Excel中得到的结果比较，检查内部均值参数是否正确设置。

```
    assert_array_equal(mean_discrete.mean, np.array([13.5, 15.5]))
```

运行转换函数，创建被转换过后的数据集。准备好测试数据（在Excel中创建的）。

```
    X_transformed = mean_discrete.transform(X_test)
    X_expected = np.array([[ 0,  0],
                           [ 0, 0],
                           [ 0, 0],
                           [ 0, 0],
                           [ 0, 0],
                           [ 1, 1],
                           [ 1, 1],
                           [ 1, 1],
                           [ 1, 1],
                           [ 1, 1]])
```

最后，测试转换器返回结果是否与期望结果一致。

```
assert_array_equal(X_transformed, X_expected)
```

调用函数，开始测试！

```
test_meandiscrete()
```

如果没弄错，上述代码应该没有问题！可以改动测试数据，用几个错误值，看看是否测试失败。要让测试结果通过，记得再改回正确的值。

如果要进行多个测试，最好使用nose测试框架来运行测试。

5.4.4　组装起来

现在，我们有了一个经过测试的转换器，拿出来用用吧。用上我们之前学到的知识，创建一个流水线，第一步使用MeanDiscrete转换器，第二步使用决策树分类器，然后进行交叉检验，输出结果。来看下代码：

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([('mean_discrete', MeanDiscrete()),
  ('classifier', DecisionTreeClassifier(random_state=14))])
  scores_mean_discrete = cross_val_score(pipeline, X, y,
    scoring='accuracy')
  print("Mean Discrete performance:
{0:.3f}".format(scores_mean_discrete.mean()))
```

正确率为0.803，结果没有之前高，对于简单的二值特征而言还算不错。

5.5　小结

本章重点为特征和转换器，以及如何将其用到数据挖掘流水线中。我们探讨了什么是好特征，以及如何用算法从数据集中选取最佳特征。然而，创建好的特征比起科学更像是一门艺术，通常需要领域相关的知识和经验。

我们创建了自己的转换器，接口与scikit-learn常用工具接口一致。后续章节会创建更多的转换器，可以使用现有函数对其进行充分

测试。

下一章将介绍文本的特征抽取方法。对于文本而言，转换器和特征类型都不少，但它们各有利弊。

第 6 章　使用朴素贝叶斯进行社会媒体挖掘

不论是书、历史文档、社会媒体、电子邮件还是其他以文字为主的通信方式，都包含大量信息。从文本数据集抽取特征，用于分类不是件容易事。然而，人们还是总结出了文本挖掘的通用方法。

本章介绍如何用强大却出奇简单的朴素贝叶斯算法消除社会媒体用语的歧义。朴素贝叶斯算法在计算用于分类的概率时，为简化计算，假定各特征之间是相互独立的，因此名字中含有朴素二字。它稍经扩展就能用于对其他类型数据集进行分类，且不依赖于数值特征。本章创建的模型在多种数据集上表现还不错，可作为很多文本挖掘研究的基准。

本章主要涉及如下内容。

- 用社交网络的API下载数据
- 用于处理文本的转换器
- 朴素贝叶斯分类器
- 用JSON保存和加载数据集
- 用NLTK库从文本中抽取特征
- 用F值评估分类效果

6.1　消歧

文本通常被称为无结构格式，虽然它包含很多信息，但是却没有标题、特定格式，句法松散，以及其他问题，导致难以从中提取有用信息。数据间联系紧密，行文中经常相互提及，交叉引用现象也很常见——从这种格式中提取信息难度很大！

我们通过比较书中的信息和大型数据库中的信息，来看下两者都有哪些方面的不同。书中有角色、主题、场所等大量信息。然而，只有去读书，并且光读还不行，更重要的是理解后才能获得书中的信息。而

位于服务器数据库的数据，每一列都有名字，都有指定的数据类型。所有的信息都在数据库中，解释起来很容易。描述数据类型、含义的信息叫作元数据，文本中缺乏这类数据。书中的目录和索引虽含有部分元数据，但是比起数据库对于数据精确的定义和描述，这点元数据实在微不足道。

文本挖掘的一个难点来自于歧义，消除歧义常被简称为消歧。当人们使用bank1这个词时，他透露的是金融相关的信息还是环境相关的（比如河岸）？在很多情况下，对我们自己来说，消除歧义比较容易（有时也不简单），但是对计算机来说难度要大得多。

> 1在英语中，bank既可以指银行，也可以指河岸。——译者注

本章将探讨如何区别Twitter消息中Python的意思。Twitter网站上的一条消息叫作tweet，它最多不能超过140个字符。这也就表明上下文信息较少。此外，没有多少元数据，虽然“#”号经常用来表示消息的主题。

当人们提及Python时，他们谈论的可能是：

- 编程语言Python
- 经典喜剧剧团Monty Python2
- 蟒蛇
- 鞋子品牌Python

> 2编程语言Python的名字正是来自于这个英国喜剧剧团的名字，Guido van Rossum很喜欢该剧团的演出，于是把他创立的编程语言命名为Python。——译者注

可能还有很多其他东西也叫Python。我们实验的目的是根据消息的内容，判断消息中的Python是不是指编程语言。

6.1.1　从社交网站下载数据

接下来，从Twitter网站下载一些语料，从中剔除垃圾信息后，用于分类任务。Twitter提供了从他们服务器采集信息的强大API，小规模使用免费，但如果你打算将Twitter数据用于商业用途，请了解下相关规定。

首先，你需要一个Twitter账号（免费）。如果没有的话，请访问http://twitter.com进行注册。

每分钟的请求数不能超过Twitter所规定的上限。写作本书时，上限为每小时180次。这个规定不好遵照执行，因此，强烈建议使用现成的库与Twitter的API进行通信。

访问Twitter数据时需要提供密钥。打开http://twitter.com，登录Twitter。

登录后，访问https://apps.twitter.com/，点击Create New App（创建新应用）。

指定新应用的名称，填好描述及要在哪个网站中使用。如果不打算在网站中使用，请在Website文本框中随意输入些内容，确保提交时表单能通过验证。Callback URL文本框空着不填——我们用不到。在下面的开发者协议处选中Yes, I agree前面的复选框（如果你确实同意），点击Create your Twitter application。

创建新应用后，先别急着关掉当前页面——后面会用到页面上的access keys（访问密钥）。接下来，需要找一个与Twitter通信的Python第三方库。选择有很多，我喜欢用Twitter官方提供的`twitter`库。

> 如果你习惯用`pip`安装第三方包，可以使用`pip3 install twitter`安装`twitter`库。如果你用的是其他系统，安装方法请参考文档https://github.com/sixohsix/twitter。

创建新的IPython笔记本文件来编写下载Twitter消息的代码。本章将创建几个不同的笔记本文件，用于不同的处理任务，所以最好是新建一个文件夹，把它们放到一起。第一个笔记本文件ch6_get_twitter专门用来下载新的Twitter语料。

首先，导入`twitter`库，设置授权令牌。刚才没让你关闭的那一页的Keys and Access Tokens（密钥和访问令牌）选项卡下，有consumer key（用户密钥）和consumer secret（请求令牌）。点击在同一页的Create my access token（创建我的访问令牌）按钮，获取访问令牌。用你得到的密钥和令牌替换下面代码中的占位符。

```
import twitter
consumer_key = "<Your Consumer Key Here>"
```

```
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token, access_token_secret,
consumer_key, consumer_secret)
```

我们将用`twitter`库提供的搜索函数（`search`）查找包含单词“Python”的消息。创建阅读器对象，提供授权信息连接到`Twitter`，然后使用阅读器对象进行搜索。在笔记本文件中，指定消息的存储位置。

```
import os
output_filename = os.path.join(os.path.expanduser("~"),
  "Data", "twitter", "python_tweets.json")
```

保存数据时要用到`json`库。

```
import json
```

接着，创建用来从Twitter网站读取数据的对象，指定授权信息，需要用到前面创建的授权对象。

```
t = twitter.Twitter(auth=authorization)
```

打开输出文件，等待写入数据，写入模式指定为“a”，这样每次在文件末尾追加新数据。使用建立好的连接，搜索单词“Python”，对于`search`方法返回的数据，我们只需要“statuses”部分。下面代码获取到Twitter消息，使用`json`库的dump函数将其转换为字符串形式后，写入到输出文件中。写完每条消息后，再写一行空行，便于把每条消息区分开来。

```
with open(output_filename, 'a') as output_file:
    search_results = t.search.tweets(q="python", count=100)['statuses']
    for tweet in search_results:
        if 'text' in tweet:
            output_file.write(json.dumps(tweet))
            output_file.write("\n\n")
```

上述循环中，还检测了消息是否包括“text”键。并不是Twitter返回的

所有对象都是消息（有些可能是用来删除消息或其他内容的动作）。消息对象与其他对象的关键不同在于消息对象中含有键“text”，这也正是我们用if语句进行检测的。

上述代码运行几分钟后，输出文件中就能有100条消息。

> 你也可以让它多运行几分钟，抓取更多的消息，需要注意的是，如果Twitter用户没有发布新消息的话，短时间内多次抓取，得到的消息很可能会有重复的。

6.1.2　加载数据集并对其分类

完成消息采集后，我们拿到了原始数据集，对它里面的消息逐条标注后，才能用于分类任务。在笔记本文件中创建一个表单，方便标注。

消息存储格式近似于JSON。JSON格式不会对数据强加过多用于表示结构的信息，可以直接用JavaScript（JSON名字也就是这么来的，JavaScript Object Notation，JavaScript对象表示法）语言读取。JSON定义了诸如数字、字符串、数组、字典等基本对象，适合存储包含非数值类型的数据。如果数据集全都是数值类型，为了节省空间和时间，最好使用类似于numpy矩阵这样的格式来存储。

我们的数据集格式和真正的JSON对象的关键区别在于，每两条消息之间有一行空行。这样做的目的是，防止新追加的消息和前面的消息混在一起（在真正的JSON格式中，追加数据没这么简单）。我们的数据集中，每两条用JSON字符串表示的消息之间有一行空行。

我们可以使用json库解析数据集，但是要先根据空行把读进来的文件拆分（split方法）成一个列表，得到真正的消息对象。

新建一个笔记本文件（我把它命名为ch6_label_twitter），指定数据集的名称。该名称即为上节指定的输出文件的名字。我们还指定用于存放每条消息所属类别的文件名。代码如下：

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_classes.json")
```

上面已经提到过，我们要使用json库，先来导入它。

```
import json
```

创建列表，用于存储从文件中读进来的每条消息。

```
tweets = []
```

遍历文件中的每一行数据。我们对不包含消息的空行（它们用于分隔消息）不感兴趣，因此，检测当前行（去除任意空白字符后的）长度是否为0。如果为0，忽略当前行，继续判断下一行。如果不为0，使用json.loads（将JSON字符串转换为Python对象）方法加载消息，将它添加到tweets列表中。代码如下：

```
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line))
```

我们现在想知道一条消息是否和我们相关（在这里，相关表示指的是编程语言Python）。接下来，在笔记本文件中嵌入HTML代码，实现JavaScript和Python之间的通信，用网页形式展示消息，便于标注。

我们要实现以下功能，向用户（你）展示一条消息，要求用户输入类别：相关还是不相关。程序保存输入结果，继续展示下一条待标注的消息。

首先，创建用于存储类别（标注结果）的列表。不管消息是不是与编程语言Python相关，我们都保存它的类别，分类器从这两类数据中学习如何预测一条消息的类别。

我们还要检测是不是有部分消息已经标注过类别了，有的话就加载这些类别。如果你标注到一半，临时有事要关闭笔记本文件，有了该功能，再打开时，代码就会加载已有类别。一般来说，对于类似的任务，考虑如何保存中间结果很有必要。这样即使计算机中途死机，努力一个小时得来的工作成果也不至于全部丢失！代码如下：

```
labels = []
if os.path.exists(labels_filename):
```

```
    with open(labels_filename) as inf:
        labels = json.load(inf)
```

接下来，创建一个简单的函数，用来返回下一条需要标注的消息。我们找到并返回第一条没有标注类别的消息即可。代码如下。

```
def get_next_tweet():
    return tweet_sample[len(labels)]['text']
```

我们实验的下一个步骤是收集用户（你！）对每条消息中的“Python”是否指的是编程语言的看法。在笔记本文件中，仅使用Python，无法通过直接与用户进行交互的方式来收集他们的反馈。因此，我们只好使用一点JavaScript和HTML代码来获取用户输入。

接下来，在笔记本中创建JavaScript程序来收集输入。可以借助魔术方法（magic function）在笔记本中直接嵌入HTML和JavaScript代码等。在笔记本的新格子中输入如下代码。

```
%%javascript
```

这样就表示下面为JavaScript代码，因此，大括号就要登场了。别担心，我们很快就会再回到Python的。请注意下面JavaScript代码必须与魔术方法`%%javascript`在同一格子里。

从下面定义的第一个JavaScript函数，就能看到在笔记本中，实现JavaScript和Python之间的通信是多么容易。这个函数的功能是向`labels`列表（Python代码中）添加一条消息所属的类别。具体做法是先加载IPython内核（kernel）对象，再用它来执行Python命令。代码如下：

```
function set_label(label){
    var kernel = IPython.notebook.kernel;
    kernel.execute("labels.append(" + label + ")");
    load_next_tweet();
}
```

函数最后调用`load_next_tweet`函数，加载下一条未标注的消息。

load_next_tweet这个JavaScript函数内部执行Python代码的原理跟上面所讲的相同：用加载的IPython内核来执行Python命令（调用前面定义的get_next_tweet函数）。

然而，获取并展示消息有点困难，需要用到回调函数，返回数据时调用该函数。回调函数的定义方法超出了本书的范围。如果你对更高级的JavaScript/Python交互方法感兴趣，请参考IPython文档。

代码如下：

```
function load_next_tweet(){
   var code_input = "get_next_tweet()";
   var kernel = IPython.notebook.kernel;
   var callbacks = { 'iopub' : {'output' : handle_output}};
   kernel.execute(code_input, callbacks, {silent:false});
}
```

回调函数叫作handle_output，下面就来创建它。当kernel.execute()调用的Python函数返回结果后，就会调用回调函数。如上所述，回调函数的详细内容不在本书讲解范围之列，我们这里用它来处理返回的纯文本格式数据，把这些数据置于表单的#tweet_text div中展示，我们随后会编写相关HTML代码。回调函数代码如下：

```
function handle_output(out){
   var res = out.content.data["text/plain"];
   $("div#tweet_text").html(res);
}
```

HTML代码中，最外层是id为tweetbox的div，它里面包着id为tweet_text的div元素用于显示下一条待标注的消息。我们还创建文本框，捕获输入的按键（否则，笔记本程序将会捕获它，JavaScript也就无法获得用户的输入）。这样我们就可以用键盘来设置消息的类别为1或0，比起用鼠标点击按钮进行选择要快——假如我们至少需要标注100条消息。

运行JavaScript代码所在的格子，就会在页面中嵌入JavaScript代码，虽然在结果区域看不到任何变化。

接着，我们来使用另一个魔术方法%%html。毫无疑问，它是用来直接在笔记本中嵌入HTML代码。在新格子中输入如下代码。

```
%%html
```

接下来，在这个格子中输入HTML代码和几行JavaScript代码。首先，定义一个div元素，用来显示当前要标注的消息。我还添加了几行标注说明。接着，创建id为tweet_text的div元素，用来显示下一条待标注的消息。如前所述，我们还需要创建文本框用来捕获按键。代码如下：

```
<div name="tweetbox">
    Instructions: Click in textbox. Enter a 1 if the tweet is
relevant, enter 0 otherwise.<br>
Tweet: <div id="tweet_text" value="text"></div><br>
<input type=text id="capture"></input><br>
</div>
```

先不要运行这个格子！

创建完表单后，我们来编写捕获键盘按键的JavaScript代码，这段脚本须写到上面HTML代码的后面，因为#tweet_text元素在HTML代码运行前在页面上还不存在。这里要用到jQuery库（IPython笔记本文件使用了该库，无需再次引入）的一个函数，当在#capture文本框元素上发生按键事件时，调用指定的函数。然而，请注意，这个格子使用的魔术方法是%%html，在这里面写JavaScript代码，需要将其放到<script>标签中。

我们只关注按键0或1，因为消息只可能属于这两个类别。通过检测存储在e.which中的ASCII码值，就能确定到底是哪个键被按下了。如果用户按下0或1，我们把类别添加到labels列表中，然后清空文本框的值。代码如下：

```
<script>
$("input#capture").keypress(function(e) {
if(e.which == 48) {
    set_label(0);
    $("input#capture").val("");
}else if (e.which == 49){
    set_label(1);
    $("input#capture").val("");
  }
});
```

忽略其他按键。

下面是本章最后几行JavaScript代码（我向你保证），我们调用load_next_tweet()函数。设置第一条待标注的消息，闭合script标签。代码如下：

```
load_next_tweet();
</script>
```

在笔记本中，运行该格子，页面上会出现HTML文本框，旁边就是第一条消息。点击文本框，如果该消息与我们相关（消息中的**Python**指的是编程语言），输入1；反之，输入0。完成后，加载下一条消息。输入类别，接着加载下一条。重复以上过程，直到标注完所有数据。

完成标注后，把所有的类别信息输出到前面定义好的类别文件中。

```
with open(labels_filename, 'w') as outf:
    json.dump(labels, outf)
```

即使没有完成标注，也可以运行上述代码，保存标注过的类别。再次运行笔记本将会加载你没有标注的消息，这样你就能继续标注。

标注消息要花点时间！消息很多的话，更是如此。如果你为了赶时间，可以下载我标注好的数据集。

6.1.3　Twitter数据集重建

数据挖掘过程会用到很多变量，它们不仅出现在挖掘算法里，数据采集等过程中也少不了它们的身影。重现实验结果很重要，因为它有助于验证或改善实验效果。[3]

> 3这一段话，照直翻译过来，总觉得缺了点什么，推测作者想要表达的意思是控制每次实验中变量的取值，确保实验结果的可比较性。——译者注

> 算法X在一个数据集上取得80%的正确率，算法Y在另一个数据集上取得90%的正确率，不能说明算法Y优于X。只有在相同的测试集上，且在相同的条件下进行测试，才能比较算法的优劣。

你用上面的代码采集到的数据集与我的不同。主要原因在于，采集时间不同，Twitter返回的搜索结果也就不同。此外，标注结果也可能会

有所差异。虽然有些消息显然是与编程语言Python有关，但也不乏模棱两可的情况，比如用我不懂的外语发布的消息，我给出的标注结果就可能有误。对于这种情况，Twitter提供了设置语言种类的接口，但即使使用该接口，返回的消息也不一定全都是你预先指定的语言。

出于这么多原因，对于从社交网站采集到的数据集重复进行相同的实验困难重重，Twitter也不例外。此外，Twitter明确禁止直接分享数据集。

解决方法之一是只分享消息编号，它是可以自由传播的。本节首先创建可以自由分享的消息编号数据集。然后，再来看下如何根据编号下载消息，重建先前的数据集。

首先，我们得把现有消息的编号及其类别保存下来。创建一个新的笔记本文件，指定接下来要用到的几个文件名。代码跟之前类似，只不过多了一个用来保存消息编号及其类别的文件。代码如下：

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
"twitter", "python_classes.json")
replicable_dataset = os.path.join(os.path.expanduser("~"),
  "Data", "twitter", "replicable_dataset.json")
```

加载消息和类别，就跟我们在上一个笔记本文件中做的那样。

```
import json
tweets = []
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line))
if os.path.exists(labels_filename):
    with open(classes_filename) as inf:
        labels = json.load(inf)
```

同时遍历所有的消息及消息所属的类别，创建新数据集，将其保存到列表中。

```
dataset = [(tweet['id'], label) for tweet, label in zip(tweets,
  labels)]
```

最后，把结果保存到文件中。

```
with open(replicable_dataset, 'w') as outf:
    json.dump(dataset, outf)
```

有了消息的编号和类别，我们就可以重建数据集。如果你想重建我在本章使用的数据集，所需代码请见本书配套代码包。

加载之前的数据集不难，但是要花些时间。新建一个笔记本文件，像之前那样指定好用于存储数据集、消息类别和消息编号的文件。我调整了下文件名，防止你覆盖掉之前采集的数据集，文件名你可以随意起，不必跟我的一样。代码如下：

```
import os
tweet_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "replicable_python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "replicable_python_classes.json")
replicable_dataset = os.path.join(os.path.expanduser("~"),
  "Data", "twitter", "replicable_dataset.json")
```

使用JSON从文件中加载消息编号及类别数据。

```
import json
with open(replicable_dataset) as inf:
    tweet_ids = json.load(inf)
```

保存所有消息的类别很容易。遍历数据集，抽取编号，用两行代码就能搞定（打开文件和保存消息）。然而，我们无法确定之后能再次获取到所有消息（例如，上次采集后，有些消息被设置为隐私），因此类别和消息可能就无法对应。

为了举例说明，我在采集数据后的第一天尝试重建数据集，却发现有两条消息已经从线上消失（可能被用户删除或设置为隐私）。因此，只输出我们实际能用到的类别就显得尤为重要。具体做法是，首先，创建`actual_labels`列表存储我们能够再次从Twitter网站获取到的消息的类别。然后，创建字典，为消息的编号和类别建立起映射关系。

代码如下：

```
actual_labels = []
label_mapping = dict(tweet_ids)
```

接下来，用twitter库根据消息编号采集消息。这可能要花点时间。导入前面用过的twitter库，创建授权令牌，用它来初始化twitter对象。

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token, access_token_secret,
  consumer_key, consumer_secret)
t = twitter.Twitter(auth=authorization)
```

遍历并抽取所有的消息编号。

```
all_ids = [tweet_id for tweet_id, label in tweet_ids]
```

然后，打开输出文件，保存消息。

```
with open(tweets_filename, 'a') as output_file:
```

Twitter API允许我们一次只能获取100条消息。因此，每次遍历100条消息。

```
    for start_index in range(0, len(tweet_ids), 100):
```

把这一批次的100个编号用逗号连接起来，便于下面使用Twitter的API根据编号查找消息。

```
        id_string = ",".join(str(i) for i in
          all_ids[start_index:start_index+100])
```

接着，调用Twitter定义的statuses/lookup方法，传入一批消息编

号（已转换为字符串），以采集这些消息。

```
        search_results = t.statuses.lookup(_id=id_string)
```

我们把返回结果中的每一条消息，按照之前采集数据集的做法，将它们依次保存到文件中。

```
        for tweet in search_results:
            if 'text' in tweet:
                output_file.write(json.dumps(tweet))
                output_file.write("\n\n")
```

最后一步（仍然属于if模块），还需要保存当前遍历到的消息的类别。获取消息类别要用到之前创建的label_mapping字典，根据消息编号查找即可。代码如下：

```
                actual_labels.append(label_mapping[tweet['id']])
```

运行上述代码，采集所有消息。如果你的数据集很大，花费时间相应较多——Twitter会限制请求的频次。最后一步，保存actual_labels到类别文件4里。

> 4再次提醒下它里面存放的是能够再次抓取到的消息的类别。——译者注

```
with open(labels_filename, 'w') as outf:
    json.dump(actual_labels, outf)
```

6.2　文本转换器

数据集创建好后，怎样在它上面施展数据挖掘的威力呢？

文本数据集包括图书、文章、网站、手稿、代码以及其他形式的文本。我们目前所见过的所有算法不是用来处理数值型就是类别型特征，怎样才能把文本转换成算法可以处理的形式？

多种测量方法能够帮上忙。比如，平均词长和平均句长可用来预测文本的可读性。除此之外，还有很多其他类型的特征，比如我们接下来

要用到的单词是否出现（word occurrence）。

6.2.1　词袋

一种最简单却非常高效的模型就是只统计数据集中每个单词的出现次数。我们来创建一个矩阵，每一行表示数据集中的一篇文档，每一列代表一个词。矩阵中的每一项为某个词在文档中的出现次数。

下面这段文字节选自托尔金（J. R. R. Tolkien）的《指环王》。

Three Rings for the Elven-kings under the sky,

Seven for the Dwarf-lords in halls of stone,

Nine for Mortal Men, doomed to die,

One for the Dark Lord on his dark throne

In the Land of Mordor where the Shadows lie.

One Ring to rule them all, One Ring to find them,

One Ring to bring them all and in the darkness bind them.

In the Land of Mordor where the Shadows lie.

—*J.R.R. Tolkien's epigraph to The Lord of The Rings*

单词the在引文中出现了9次，单词in、for、to和one各出现了4次。单词ring和of各出现3次。

从中选取几组数据，创建一个简单的数据集。

		单词			
频次					

可以用`counter`方法统计列表中各字符串出现次数。统计单词时，通常将所有字母转换为小写，因此定义字符串时顺便把它转换为小写形式。代码如下：

```
s = """Three Rings for the Elven-kings under the sky,
```

```
Seven for the Dwarf-lords in halls of stone,
Nine for Mortal Men, doomed to die,
One for the Dark Lord on his dark throne
In the Land of Mordor where the Shadows lie.
One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them.
In the Land of Mordor where the Shadows lie. """.lower()
words = s.split()
from collections import Counter
c = Counter(words)
```

`c.most_common(5)`输出出现次数最多的前5个词，竟然有4个词并列第二，多输出几个就能看出词频的差异了。

词袋模型主要分为以下三种：第一种像上面这样使用词语实际出现次数作为词频。缺点是当文档长度差异明显时，词频差距会非常大。第二种是使用归一化后的词频，每篇文档中所有词语的词频之和为1。这种做法优势明显，它规避了文档长度对词频的影响。第三种，直接使用二值特征来表示——单词在文档中出现值为1，不出现值为0。本章使用第三种。

另外一种（更）通用的规范化方法叫作词频—逆文档频率法（term frequency-inverse document frequency，简写为tf-idf），该加权方法用词频来代替词的出现次数，然后再用词频除以包含该词的文档的数量。第10章将会用到词频—逆文档频率法。

Python有很多用于处理文本的库。我们将使用主流的**NLTK**库（Natural Language ToolKit，自然语言处理工具集）抽取特征。`scikit-learn`提供进行类似处理的`CountVectorizer`类，建议你花点时间了解下（第9章将会用到）。然而在分词方面，NLTK提供更多选择。如果你打算用Python做自然语言处理，NLTK是个不错的选择。

6.2.2　N元语法

比起用单个词作特征，使用**N**元语法能更好地描述文档，具体优势稍后会讲。N元语法是指由几个连续的词组成的子序列。拿我们的数据集来讲，N元语法指的是每条消息里一组连续的词。

N元语法的计算方法跟计算单个词语方法相同，我们把构成N元语法的几个词看成是词袋中的一个词。数据集5中每一项就变成了N元语法在给定文档中的词频。

5指用N元语法作为特征的数据集。——译者注

N元语法中的参数n，对于英语这门语言，一开始取2到5之间的值就可以，有些应用可能要使用更高的值

举个例子吧，当 n 取3时，我们从下面引文中抽取前几个N元语法。

Always look on the bright side of life.

第一个N元语法（三元）是Always look on，第二个是look on the，第三个是on the bright。你可能已经发现，几个N元语法有重合，其中三个词有不同程度的重复。

N元语法比起单个词有很多优点。这个简单的概念不用通过大量的计算，就提供了有助于理解词语用法的上下文信息。它的缺点是特征矩阵变得更为稀疏——一个N元语法不太可能出现两次（尤其是在Twitter消息及其他短文本中！）。

对于社会媒体所产生的内容以及其他短文档，N元语法不可能出现在多篇不同的文档中，除非是转发。然而，在长文档中，N元语法就很有效。

文档的另外一种N元语法关注的不是一组词而是一组字符（虽然字符N元语法6有多种计算方法！）。字符N元语法有助于发现拼写错误，除此之外，还有其他好处。本章及第9章都将测试这种N元语法的效果。

6英文为Character N-gram。 ——译者注

6.2.3　其他特征

除了N元语法外，还可以抽取很多其他特征，其中就包括句法特征，比如特定词语在句子中的用法。对于需要理解文本含义的数据挖掘应用，往往会用到词性。本书限于篇幅将不涉及这些特征。如果你对此感兴趣，推荐你看由Packt出版的*Python 3 Text Processing with NLTK 3 Cookbook*，作者为Jacob Perkins。

6.3　朴素贝叶斯

毫无疑问，朴素贝叶斯概率模型是以对贝叶斯统计方法的朴素解释为基础。尽管存在朴素的一面，这种方法应用面很广且都取得了不错的效果。特征类型和形式多样的数据集也可以用它进行分类，本章重点讲解如何用二值化后的特征组成的词袋模型来分类。

6.3.1　贝叶斯定理

大多数人在开始学习统计学时，都会被灌输从频率论者角度出发看问题的思想，即假定数据遵从某种分布，我们的目标是确定该种分布的几个参数。我们假定参数是固定的（也可能不正确），然后用自己的模型来套这些数据，甚至通过测试来证明数据与我们的模型相吻合。

相反，贝叶斯统计实际上是根据普通人（非统计学家）实际的推理方式来建模。我们用拿到的数据，来更新模型对某事件即将发生的可能性的预测结果。在贝叶斯统计学中，我们使用数据来描述模型，而不是使用模型来描述数据，用数据证实拍脑瓜得出的模型是典型的频率论者的做法。

贝叶斯定理旨在计算P(A | B)的值，也就是在已知B发生的条件下，A发生的概率是多少。大多数情况下，B是被观察事件，比如“昨天下雨了”，A为预测结果“今天会下雨”。对数据挖掘来说，B通常是观察样本个体，**A**为被预测个体所属类别。下一节将学习如何在数据挖掘领域使用贝叶斯定理。

贝叶斯定理公式如下：

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

举例说明，我们想计算含有单词drugs的邮件为垃圾邮件的概率（正如我们认为含有该词的Twitter消息可能是兜售药品的垃圾广告）。

在这里，A为“这是封垃圾邮件”。7我们先来计算P(A)，它也被称为先验信念（prior belief）。计算方法是，统计训练集中垃圾邮件的比例。如果我们的数据集每100封邮件有30封垃圾邮件，P(A)为30/100或0.3。

> 7原文直译为“A为这是封垃圾邮件的概率”，其实此处A表示的是事件，而不是概率。——译者注

B表示“该封邮件含有单词drugs”。类似地，我们可以通过计算数据集

中含有单词drugs的邮件数量得到P(B)。如果每100封邮件中有10封邮件包含单词drugs，那么P(B)就为10/100或0.1。计算P(B)时，我们不关注邮件是不是垃圾邮件。

P(B|A)指的是垃圾邮件中含有单词drugs的概率，计算起来也很容易，统计训练集中所有垃圾邮件的数量以及其中含有单词drugs的数量。30封垃圾邮件中，如果有6封含有单词drugs，那么P(B|A)就为6/30或0.2。

现在，我们根据贝叶斯定理就能计算出P(A|B)，得到含有drugs的邮件为垃圾邮件的概率。把上面求出来的各项代入前面的贝叶斯公式，得到结果0.6。这表明如果邮件中含有drugs这个词，那么该邮件为垃圾邮件的概率为60%。

请注意上述例子的实证性特点——我们使用的经验或证据直接来自于数据集，而不是事先假定好的某种分布。相比之下，频率论方法会要求我们首先为训练集中的单词出现概率创建某种分布形式。

6.3.2　朴素贝叶斯算法

回过头看下贝叶斯公式，我们可以用它计算个体从属于给定类别的概率。因此，它可以用来分类。

我们用C表示某种类别，用D表示数据集中一篇文档，来计算贝叶斯公式所要用到的各种统计量，对于不好计算的，做出朴素假设，简化计算。朴素贝叶斯分类算法使用贝叶斯定理计算个体从属于某一类别的概率。

P(C)为某一类别的概率，可以从训练集中计算得到（方法跟上文检测垃圾邮件例子所用到的一致）。统计训练集所有文档从属于给定类别的百分比。

P(D)为某一文档的概率，它牵扯到各种特征，计算起来很困难，但是在计算文档属于哪个类别时，对于所有类别来说，P(D)相同，因此根本就不用计算它。稍后我们来看下怎么处理。

P(D|C)为C类含有文档D的概率。由于D包含多个特征，计算起来可能很困难，这时朴素贝叶斯算法就派上用场了。我们朴素地假定各个特征之间是相互独立的，分别计算每个特征（D1、D2、D3等）在给定类别出现的概率，再求它们的积。



```
P(D|C) = P(D1|C) x P(D2|C).... x P(Dn|C)
```

上式右侧对于二值特征相对比较容易计算。直接在数据集中进行统计，就能得到所有特征的概率值。

相反，如果我们不做朴素的假设，就要计算每个类别不同特征之间的相关性。这些计算很难完成，如果没有大量的数据或足够的语言分析模型也不可能完成。

到这里，算法就很明确了。对于每个类别，我们都要计算P(C|D)，忽略P(D)项。概率较高的那个类别即为分类结果。由于P(D)对每个类别来说都是相等，去掉它对最终预测结果没有多大影响。

6.3.3　算法应用示例

举例说明下计算过程，假如数据集中有以下一条用二值特征表示的数据：[1, 0, 0, 1]。

训练集中有75%的数据属于类别0，25%属于类别1，且每个特征属于每个类别的似然度如下。

类别0：[0.3,　0.4,　0.4,　0.7]

类别1：[0.7,　0.3,　0.4,　0.9]

拿类别0中特征1的似然度举例子，上面这两行数据可以这样理解：类别**0**中有**30%**的数据，特征**1**的值为**1**。

我们来计算一下这条数据属于类别0的概率。类别为0时，P(C＝0) ＝ 0.75。

朴素贝叶斯算法用不到P(D)，因此我们不用计算它。我们来看下计算过程。

```
P(D|C=0) = P(D1|C=0) x P(D2|C=0) x P(D3|C=0) x P(D4|C=0)
= 0.3 x 0.6 x 0.6 x 0.7
= 0.0756
```

第二、三个值为0.6，是因为在上面我们给出的那条数据（[1, 0, 0, 1]）中，这两个特征的值为0。而我们给出的似然度表示特征值取1时，在各类别的

概率。因此，特征值为0的概率为：P(0) ＝ 1 – P(1)
。

现在，我们就可以计算该条数据从属于每个类别的概率。需要提醒的是，我们没有计算P(D)，因此，计算结果不是实际的概率。由于两次都不计算P(D)，结果具有可比较性，能够区分出大小就足够了。来看下计算结果。

```
P(C=0|D) = P(C=0) P(D|C=0)
= 0.75 * 0.0756
= 0.0567
```

接着，计算类别1的概率。

```
P(C=1) = 0.25
```

朴素贝叶斯公式不需要计算P(D)。我们来看下计算过程。

```
P(D|C=1) = P(D1|C=1) x P(D2|C=1) x P(D3|C=1) x P(D4|C=1)
= 0.7 x 0.7 x 0.6 x 0.9
= 0.2646
P(C=1|D) = P(C=1)P(D|C=1)
= 0.25 * 0.2646
= 0.06615
```

通常，P(C＝0|D) ＋ P(C＝1|D)应该等于1。毕竟，只有这两种选择！然而，我们这里二者的和不为1，因为我们在计算时省去了公式中的P(D)项。

该条数据应该被分到类别1中。计算过程中，你可能已经猜到结果了。看到最终结果两个类别的概率如此接近，你可能还是会有点惊讶。毕竟，类别为0时，P(D|C)的概率比类别为1时高很多。这是因为我们给出的先验概率很高，即大部分数据都属于类别0。

如果训练集中两类数据数量相同，结果就会大为不同。假设P(C＝0)、P(C＝1) 各为0.5，再计算下结果看看。

6.4　应用

接下来，创建流水线，接收一条消息，仅根据消息内容，判断它是否

与编程语言Python相关。

我们使用NLTK抽取特征。NLTK提供了大量用于自然语言处理的工具。后续章节还会继续使用它。



用pip安装NLTK：`pip3 install nltk`

如果安装失败，请参考NLTK安装指南：www.nltk.org/install.html。

接下来，创建流水线抽取词语特征，并使用朴素贝叶斯算法对消息进行分类。流水线包括以下步骤。

(1) 用NLTK的`word_tokenize`函数，将原始文档转换为由单词及其是否出现组成的字典。

(2) 用`scikit-learn`中的`DictVectorizer`转换器将字典转换为向量矩阵，这样朴素贝叶斯分类器就能使用第一步中抽取的特征。

(3) 正如前几章做过的那样，训练朴素贝叶斯分类器。

(4) 还需要新建一个笔记本文件ch6_classify_twitter（本章最后一个），用于分类。

6.4.1 抽取特征

我们使用NLTK抽取单词是否出现作为特征。我们想在流水线中使用NLTK，但是它的接口与转换器接口不一致。因此，需要创建一个包含`fit`和`transform`方法的基础转换器，只有这样才能在流水线中使用。

首先，创建转换器类。这个类不需要进行预处理，只需要抽取特征。因此，`fit`函数不做任何操作，只返回它自身（self）即可，转换器对象要用到它。

转换器函数就有点复杂。我们要用它从每篇文档中抽取单词，如果单词出现，记为`True`。注意这里使用二值特征，单词在文档中出现，值为`True`，反之，值为`False`。如果我们想使用词频，就要创建用来统计单词频率的字典，前几章讲过。

来看下代码。

```
from sklearn.base import TransformerMixin
from nltk import word_tokenize

class NLTKBOW(TransformerMixin):
    def fit(self, X, y=None):
        return self
    def transform(self, X):
        return [{word: True for word in word_tokenize(document)}
                 for document in X]
```

返回结果为一个元素为字典的列表，第一个字典的各项为第一条消息中的所有词语。字典的每一项用单词作为键，值为True，表示该词在该条消息中出现过。字典中没有出现的词，表示这条消息里不包含该词。当然，我们也可以用值False来表示没在消息中的词，但是那样太浪费存储空间，也没有必要。

6.4.2 将字典转换为矩阵

这一步是将上面得到的字典转换为可以用分类器进行处理的矩阵。在DictVectorizer的帮助下，这一步变得非常容易。

DictVectorizer类接受元素为字典的列表，将其转换为矩阵。矩阵中的各个特征为所有字典中的每个键，特征值就是特征在文本中是否出现。用代码生成字典很容易，但是实现的很多数据挖掘算法更喜欢接收矩阵格式的数据，于是DictVectorizer显得格外有用。

数据集中，每个字典用单词作为键，单词只有在对应的消息中出现，这个单词才会出现在字典里。因此，矩阵以每个单词作为特征，如果消息中出现该单词，那么相应的特征值就为True。

导入DictVectorizer后，就可以使用它。

```
from sklearn.feature_extraction import DictVectorizer
```

6.4.3 训练朴素贝叶斯分类器

最后，我们需要组装分类器，因为本章使用朴素贝叶斯算法，数据集只包含二值特征，因此我们使用专门用于二值特征分类的BernoulliNB分类器，它用起来很简单。就像是DictVectorizer

一样，我们先来导入它，把它添加到流水线中。

```
from sklearn.naive_bayes import BernoulliNB
```

6.4.4　组装起来

终于等到把所有部件组装起来的时候了。像前面做过的那样，在笔记本文件中，指定好文件名，加载数据集和类别数据。注意是消息（不是消息编号）及它们的类别所在的文件。代码如下：

```
import os
input_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "python_tweets.json")
labels_filename = os.path.join(os.path.expanduser("~"), "Data",
  "twitter", "python_classes.json")
```

加载消息。我们只对消息内容感兴趣，因此只提取和存储它们的text值。代码如下：

```
tweets = []
with open(input_filename) as inf:
    for line in inf:
        if len(line.strip()) == 0:
            continue
        tweets.append(json.loads(line)['text'])
```

加载消息的类别。

```
with open(classes_filename) as inf:
    labels = json.load(inf)
```

创建流水线，把所有部件组合起来。流水线包含以下三个部分。

- 我们创建的NLTKBOW转换器
- DictVectorizer转换器
- BernoulliNB分类器

流水线代码如下：

```
from sklearn.pipeline import Pipeline
pipeline = Pipeline([('bag-of-words', NLTKBOW()),
                     ('vectorizer', DictVectorizer()),
                     ('naive-bayes', BernoulliNB())
                     ])
```

我们几乎现在就可以运行流水线，用之前多次用过的cross_val_score方法来计算正确率。但是在这之前，我们要介绍一种比正确率更好的评价指标。我们后面会看到，每个类别数据量不同的情况下，正确率不足以说明算法的优劣。

6.4.5　用F1值评估

选择评价指标时，了解它们的适用范围很重要。正确率应用范围很广，理解起来比较容易，计算起来也方便。但是，造假很容易。换句话说，你很容易就能实现一个正确率很高，但实际用处不大的算法。

我们的消息数据集（你的可能与此不同）中，50%的消息与编程语言有关，50%不相关，很多数据集不会这么均匀（balanced）。

例如，对于垃圾邮件过滤器而言，其所处理的邮件很可能80%以上都是垃圾邮件，倘若一个过滤器把所有邮件都标为垃圾邮件，它没有实际应用价值，但是正确率却高达80%！

为了解决这个问题，我们使用另一个最为常用的评价指标**F1**值（也被称为F值、F-measure，或者其他变体**8**）。

> 8比如F0.5等。——译者注

F1值是以每个类别为基础进行定义的，包括两大概念：准确率（precision）和召回率（recall）。准确率是指预测结果属于某一类的个体，实际属于该类的比例。召回率是指被正确预测为某个类别的个体数量与数据集中该类别个体总量的比例。

在我们这里，可分别对两个类别（相关和不相关）的分类情况计算F1值。但是，我们只关注相关这一类数据的分类情况。因此，准确率计算变为以下问题：在所有被预测为相关的消息中，真正相关的占比多少？类似地，召回率就转化为：数据集所有相关的消息中，有多少被正确预测为相关的？

计算出准确率和召回率后，就能得到F1值，它是两者的调和平均数。

$$F1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

将scoring参数的值设置为F1，就能使用scikit-learn中的F1方法。默认将会返回类别为1的F1值。使用以下代码求得F1值。

```
scores = cross_val_score(pipeline, tweets, labels, scoring='f1')
```

输出平均值。

```
import numpy as np
print("Score: {:.3f}".format(np.mean(scores)))
```

结果为0.798，这表明预测含有Python的消息与编程语言有关的F1值差不多为80%。这是使用包含200条消息的数据集取得的结果。如果采集更多消息作为训练数据，你会发现结果还会提升！

更多的数据通常意味着更好的结果，但也不一定！

6.4.6 从模型中获取更多有用的特征

你可能存在这样的疑问："什么特征才是判断一条消息是否相关的最好的特征？"我们可以从朴素贝叶斯模型中抽取该信息，找到朴素贝叶斯算法所认为的最好的特征。

首先，训练得到一个新模型。使用cross_val_score在测试集上得到交叉检验结果不难，但是要得到模型本身就不容易了。为了得到模型，我们只好用流水线的fit函数，创建新模型。代码如下：

```
model = pipeline.fit(tweets, labels)
```

注意我们不是在评价模型，训练/测试集的切分就没那么严格。然而，在使用这些特征之前，应该在单独的测试集上评价其表现。为保证讲解得清楚明了，跳过这一部分。

借助流水线的named_steps属性和步骤名（创建流水线对象时，我

们自己定义的），就能访问流水线每一个步骤。例如，可以像下面这样访问朴素贝叶斯模型。

```
nb = model.named_steps['naive-bayes']
```

从这个模型中，可以抽取每个单词的对数概率，比如log(P(A|f))，其中f为给定特征。

之所以使用对数概率，是因为实际值非常小。例如，第一个值为-3.486，实际概率约为0.03。计算中涉及很小的概率值时，常使用对数概率来防止数值下溢，因为非常小的值往往会被约等于0。所有概率连乘，其中一个值为0，最终结果将为0！而实际上即使是很小的值，大小不同，对分类的贡献率也会有所差异，值越大的特征贡献率越大。

把得到的对数概率数组按照降序排列，找出最有用的特征。降序排列，需要在值前加个负号。代码如下：

```
top_features = np.argsort(-feature_probabilities[1])[:50]
```

上面代码只是给出特征索引值而没有给出实际的特征名称。这样看不出什么东西来，因此，需要把特征索引和特征名称对应起来。流水线的DictVectorizer这一步很关键，它是用来创建矩阵的。幸运的是，它也记录了特征名称和索引值的映射关系。因此，可以从流水线的这一部分抽取特征。

```
dv = model.named_steps['vectorizer']
```

通过在DictVectorizer的feature_names_属性中进行查找，找到最佳特征的名称，然后将其输出。输入下面代码，运行后将得到最佳特征列表。

```
for i, feature_index in enumerate(top_features):
    print(i, dv.feature_names_[feature_index],
      np.exp(feature_probabilities[1][feature_index]))
```

前几个特征为“:”“http”“#”“@”。结合采集的数据来判断，我们认为这些很可能是噪音（虽然冒号在编程语言之外的文本中使用的相对

较少）。更多的训练数据，有助于减少这些噪音对分类的影响。接着往下看，下面这些特征更像是与编程语言有关。

```
7 for 0.188679245283
11 with 0.141509433962
28 installing 0.0660377358491
29 Top 0.0660377358491
34 Developer 0.0566037735849
35 library 0.0566037735849
36 ] 0.0566037735849
37 [ 0.0566037735849
41 version 0.0471698113208
43 error 0.0471698113208
```

还有一些特征是在工作环境中提及Python，因此可能指的是编程语言Python。（虽然作为自由职业者的耍蛇人也可能使用Python这个词，但是他们较少活跃在Twitter上。）

```
22 jobs 0.0660377358491
30 looking 0.0566037735849
31 Job 0.0566037735849
34 Developer 0.0566037735849
38 Freelancer 0.0471698113208
40 projects 0.0471698113208
47 We're 0.0471698113208
```

上面最后一个特征通常出现在诸如“We're looking for a candidate for this job”（我们正在招聘符合该职位要求的员工）这样的消息中。

查看这些特征，我们收获不小，有了这些特征，我们自己经过训练也可以完成分类任务，寻找这些特征的共同点（与主题相关），或者去除讲不通的特征。例如，词“RT”虽然排名比较靠前，然而，这是Twitter网站表示转发的用语。经验丰富的工作人员就可以从特征列表中删除这个词，降低由于数据集很小而引入的噪音对分类结果的影响。

6.5 小结

本章研究的是文本挖掘——特征的抽取、应用及扩展方法，具体研究任务是根据语境消除词语的歧义——即一条消息中的Python是否指编程语言。我们使用Twitter提供的API下载消息，使用在笔记本文件中建立的表单完成语料标注。

我们还考虑了实验结果的可再现性。虽然Twitter不允许你把自己采集的数据给别人使用，但是消息编号是可以共享的。我们编写代码，保存消息编号，再用这些编号来重建先前的数据集。由于有些消息被删除或是出于其他原因，我们无法再次获取它们。

我们用朴素贝叶斯分类器对文本进行分类，该分类器是以贝叶斯定理为基础，它使用数据来更新模型，而不是像频率论者那样从模型出发。这有助于整合新数据到模型中，利用新数据来更新模型和使用先验知识。此外，基于各特征相互独立的朴素假设，便于计算各特征出现在给定类别的概率，而不用考虑各特征之间复杂的相关性。

我们用词语是否出现作为特征值——即消息是否含有某个词，这种模型9叫作词袋模型。虽然它丢掉了词语在句子中的位置等信息，但它在很多数据集上表现不凡。

> 9把文档看成由一个个孤立的、无前后位置关系的单词组成的。——译者注

朴素贝叶斯分类器结合词袋模型，组装成的流水线功能强大。对于大多数文本挖掘任务，它都能取得很好的效果。在尝试更高级的模型之前，用这样的分类器取得的分类结果作为参考的基准很不错。另外一个优点是，朴素贝叶斯分类器不需要调整任何参数（如果你愿意折腾的话，也确实有几个）。

下一章将介绍怎样从另外一种数据类型——图中抽取特征，尝试解决向社会媒体网站用户推荐感兴趣的人这一问题。

第 7 章　用图挖掘找到感兴趣的人

很多事物都可用图来表示，大数据、社交网络和物联网时代尤其如此。特别值得一提的是，社交网络具有很大的商业价值，像Facebook这样每天有5亿活跃用户（50%的用户每天都会登录）的网站，通过定向投放广告获利颇丰。怎样才能打造一个这样的网站呢？对于网站而言，要想留住用户，你得有用户感兴趣的人或内容。

本章将探讨相似度这个概念以及如何根据它来创建图模型，并且探讨如何使用连通分支（connected components）把大图分为有意义的小图。本章即将介绍的算法引入聚类分析概念——根据相似度，把大数据集划分为几个子集。第10章将对聚类分析做更加深入的讲解。

本章主要内容如下：

- 用社交网络数据创建图
- 加载和保存创建的分类器
- `NetworkX`库
- 图到矩阵的转换
- 距离和相似度
- 根据打分函数优化参数
- 损失函数和打分函数

7.1　加载数据集

本章的任务是根据社交网络用户的好友信息，向他们推荐好友。逻辑为：如果两个用户有共同的好友，那么这两个人相似度很高，值得向彼此推荐。

利用上一章介绍的Twitter API来获取数据，创建一个小的社交网络图。寻找喜欢同一个话题（与上一章相同，依旧是编程语言Python）的用户，从中选一部分，再获取这些人的好友列表（他们关注的人）。有了以上数据，就能根据两个用户共同拥有的好友数量，计算

他们的相似度。

> 除了Twitter，还有很多其他社交网站。之所以使用Twitter，是因为其提供的API很容易获取所需要的数据。其他网站，比如Facebook、LinkedIn和Instagram等也开放这些数据，但是获取起来难度更大些。

现在来采集数据，新建一个IPython Notebook笔记本文件，初始化`twitter`连接实例，方法与上一章相同。可以沿用上一章所创建的Twitter应用的密钥信息，再创建新的也可以：

```
import twitter
consumer_key = "<Your Consumer Key Here>"
consumer_secret = "<Your Consumer Secret Here>"
access_token = "<Your Access Token Here>"
access_token_secret = "<Your Access Token Secret Here>"
authorization = twitter.OAuth(access_token, access_token_secret,
  consumer_key, consumer_secret)
t = twitter.Twitter(auth=authorization, retry=True)
```

指定输出文件名：

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "twitter")
output_filename = os.path.join(data_folder, "python_tweets.json")
```

依然用`json`库保存数据：

```
import json
```

接下来，收集一组用户信息。像上一章那样搜索推文，查找包含单词python的消息。首先，创建两个列表，用于存储消息文本内容及相应的用户信息。稍后会用到用户编号，因此，需要创建个字典，将用户昵称和编号对应。代码如下：

```
original_users = []
tweets = []
user_ids = {}
```

搜索包含python的消息，遍历搜索结果：

```
search_results = t.search.tweets(q="python",
  count=100)['statuses']
for tweet in search_results:
```

我们只对消息感兴趣，对Twitter可能返回的其他信息不感兴趣，因此检测消息中有没有text属性：

```
    if 'text' in tweet:
```

如果有，记录用户昵称、消息正文，并且将用户昵称和编号关联起来。代码如下：

```
        original_users.append(tweet['user']['screen_name'])
        user_ids[tweet['user']['screen_name']] =
          tweet['user']['id']
        tweets.append(tweet['text'])
```

运行上述代码将会得到大约100条消息，有时可能会少些。但是，这些消息并不是全都和编程语言有关。

7.1.1　用现有模型进行分类

正如上一章所说，不是所有提到python的消息都与编程语言有关。怎么才能找出所需要的消息？上一章用过的分类器就能派上用场了。分类器虽不完美，但使用它进行分类，能够过滤掉搜索结果中相当一部分噪音。

在本案例中，只对谈论Python编程语言的用户感兴趣。使用上一章创建的分类器找出与编程语言Python相关的消息。发表这类消息的人是我们真正感兴趣的用户。具体做法如下。

首先，需要保存朴素贝叶斯分类模型。打开上一章创建分类器的IPython笔记本文件。如果已经关闭，笔记本不记得之前的操作，需要运行所有的代码片段，方法是点击笔记本的Cell菜单，选择Run All。

运行完后，选择页面下方的空白格子。如果没有，选中最后一个格子，点击Insert菜单，选择Insert Cell Below选项。

我们将使用joblib库保存并加载模型。

> joblib包含在scikitlearn库中。

首先，导入joblib库，为模型指定输出文件名（确保目录存在，否则无法创建）。我把模型保存在我自己创建的Models目录下，你也可以将其保存到其他地方。代码如下：

```
from sklearn.externals import joblib
output_filename = os.path.join(os.path.expanduser("~"), "Models",
  "twitter", "python_context.pkl")
```

接着，用joblib库的dump函数，与json库的dump函数功能类似。参数为模型本身（怕你忘了，捎带提下，模型名称为model）和输出文件名。

```
joblib.dump(model, output_filename)
```

运行上述代码，模型将会保存到指定文件中。接着，回到上小节创建的笔记本文件，加载模型。

复制下面代码，在当前笔记本文件中再次指定模型的文件名：

```
model_filename = os.path.join(os.path.expanduser("~"), "Models",
  "twitter", "python_context.pkl")
```

检查下这里的文件名是否与保存模型所用的文件名相同。接着，需要重建NLTKBOW类，因为它是定制的类，无法直接用joblib加载。更好的处理方法后续章节会讲。现在，只需从上一章的代码中复制整个NLTKBOW类及它所依赖的模块：

```
from sklearn.base import TransformerMixin
from nltk import word_tokenize

class NLTKBOW(TransformerMixin):
    def fit(self, X, y=None):
        return self

    def transform(self, X):
        return [{word: True for word in word_tokenize(document)}
                 for document in X]
```

现在，只需调用joblib的load函数就能加载模型：

```
from sklearn.externals import joblib
context_classifier = joblib.load(model_filename)
```

context_classifier与第6章的model对象功能相同，是Pipeline对象的实例，它的三个步骤也与第6章流水线（NLTKBOW、DictVectorizer和BernoulliNB分类器）相同。

调用context_classifier模型的predict函数，预测消息是否与编程语言相关。代码如下：

```
y_pred = context_classifier.predict(tweets)
```

如果第*i*条消息与编程语言有关，那么y_pred中的第*i*项为1，否则为0。据此，可以获取到所有与编程语言有关的消息及用户：

```
relevant_tweets = [tweets[i] for i in range(len(tweets)) if y_pred[i]
  == 1]
relevant_users = [original_users[i] for i in range(len(tweets)) if
  y_pred[i] == 1]
```

用我采集的数据，共找到46名相关用户。比起之前的100条消息/用户显得有点少，但还是可以以此为基础来构建社交网络。

7.1.2　获取Twitter好友信息

接下来，需要获得以上每个用户的好友。这里好友的定义是：用户正在关注的人。Twitter提供了相应API：friends/ids，它好用的地方在于一次调用最多能获得高达5000个好友的编号，不好的地方是每15分钟最多只能调用15次，这也就意味着获取每个用户的好友列表至少需要一分钟——如果好友数量多于5000，需要更多时间（这种情况比想象中的更常见）。

然而，代码相对比较简单。接下来两节还要用到获取好友列表的功能，干脆把它封装成一个函数。首先，声明该函数，接收两个参数，一个是用于连接Twitter的对象，另一个是用户编号。函数返回由第二个参数指定的用户的所有好友，创建列表friends来存储好友们的编号。由于要用到时间模块time，顺便导入它。在给出完整的函数

前，会逐一讲解每一部分代码。开始部分如下：

```
import time
def get_friends(t, user_id):
    friends = []
```

听起来有些奇怪，很多Twitter用户拥有5000个以上的好友。因此，要用到Twitter的pagination（分页）对象。Twitter使用游标管理多页数据。当向Twitter请求数据时，不仅返回所需数据，还返回数据类型为整数的游标，Twitter用游标跟踪每一次请求。如果没有更多内容，游标为0；否则，可以使用游标获得下一页数据。开始把游标设置为-1，表明是数据的开始：

```
    cursor = -1
```

接下来，只要游标不为0（当为0时，数据已采集完），就持续执行下面的循环。请求用户的好友数据，并将这些好友的编号添加到friends列表中。这里用到了try语句，请求过程中可能会遇到问题，使用try以便于处理异常。results字典中键为ids的那一项保存的是好友编号，将好友编号添加到friends列表中。然后，更新游标。下一次迭代时，游标使用刚更新过的值。最后，检查好友数量是否超过一万，如果超过，跳出循环。代码如下：

```
    while cursor != 0:
        try:
            results = t.friends.ids(user_id= user_id,
              cursor=cursor, count=5000)
            friends.extend([friend for friend in results['ids']])
            cursor = results['next_cursor']
            if len(friends) >= 10000:
                break
```

 这里有必要插入条警告信息。从互联网抓取数据，可能会时不时遇到这样那样奇怪的事情。我在编写上面这段代码时遇到的一个问题是有些用户好友异常多。为了应对这个问题，代码中增加了安全退出机制，用户好友再多，只要获取到他的一万名好友信息后就强制退出。如果想采集用户的所有好友，请删掉这两行，但是请注意碰巧遇到好友特别多的用户，会卡很长时间。

接着为可能捕获到的异常编写处理方法。最可能犯的错误是不小心达到了API访问次数的上限（虽然sleep语句可以在一定程度上避免这种情况，但在sleep语句结束前，中止代码执行后，接着在原本应该停顿的时间重新运行代码，就会出现这种情况）。这样的话，返回结果results为None，代码会抛出TypeError（类型错误）异常。遇到这种情况，等待五分钟，再次执行新一轮循环，希望Twitter能够重新计算时间，这样在接下来的15分钟就又能请求15次数据。除此之外，可能还有其他原因引发的TypeError异常。对于这种异常，需要单独处理，因此需要返回具体错误信息，方便查找错误起因。代码如下：

```
        except TypeError as e:
            if results is None:
                print("You probably reached your API limit,
                  waiting for 5 minutes")
                sys.stdout.flush()
                time.sleep(5*60) # 5 minute wait
            else:
                raise e
```

第二类异常可能发生在Twitter这一端，比如查找的用户不存在或是其他和数据相关的错误。这时，不要再尝试继续获取导致报错的用户的好友数据，返回已经得到的即可（很可能为0）。代码如下：

```
        except twitter.TwitterHTTPError as e:
            break
```

接着处理API限制。Twitter只允许每15分钟调用15次获取用户好友数据的函数，15次过后，需要等一分钟再继续进行。把这段代码放到finally语句中，遇到异常后，也会执行：

```
        finally:
            time.sleep(60)
```

函数最后返回所采集到的好友编号：

```
    return friends
```

下面是完整的函数：

```
```

```
import time
def get_friends(t, user_id):
    friends = []
    cursor = -1
    while cursor != 0:
        try:
            results = t.friends.ids(user_id= user_id,
              cursor=cursor, count=5000)
            friends.extend([friend for friend in
              results['ids']])
            cursor = results['next_cursor']
            if len(friends) >= 10000:
                break
        except TypeError as e:
            if results is None:
                print("You probably reached your API limit,
                  waiting for 5 minutes")
                sys.stdout.flush()
                time.sleep(5*60) # 5 minute wait
            else:
                raise e
        except twitter.TwitterHTTPError as e:
                break
               finally:
                   time.sleep(60)
    return friends
```

7.1.3　构建网络

现在着手构建网络。从最初的46名（你得到的数据集可能与我的有差异）用户出发，找到每个人的好友，将其保存到字典里（从user_id字典拿到用户编号）：

```
friends = {}
for screen_name in relevant_users:
    user_id = user_ids[screen_name]
    friends[user_id] = get_friends(t, user_id)
```

删除没有好友的孤家寡人。对于这些人，无法按照既定逻辑向他们推荐好友。也许可以从他们发表的消息以及关注他们的人那里发现有用线索。感兴趣的读者可自行研究，限于篇幅，本章不过多涉及。把这些人过滤掉。代码如下：

```
friends = {user_id:friends[user_id] for user_id in friends
             if len(friends[user_id]) > 0}
```

这样能剩下30到50个用户，具体数量与你之前的搜索结果有关。这些不大够用，得想办法再抓取些数据，使总用户增加到150个。下面的代码运行起来要费点时间——由于Twitter API的限制，一分钟只能取到一个用户的好友数据。不用算也知道150个用户数据，需要150分钟，也就是2.5个小时。既然要花这么多时间来采集用户的好友数据，得保证这些用户是想要的才行。

那么，什么用户才是想要的呢？考虑到要根据共同的好友向用户推荐他们可能感兴趣的人，因此就查找有共同好友的人。找到现有用户的好友，从他们中间找出在现有用户中间有更多好友的人。因此，需要统计这些用户的好友数量，即出现在已有用户的friends列表中的次数。从事数据挖掘任务时，确定采样策略前，考虑好应用的实际目标很有必要。这里向相似的用户推荐好友更可行。

具体做法为遍历所有用户的好友列表，统计每个好友的出现次数：

```
from collections import defaultdict
def count_friends(friends):
    friend_count = defaultdict(int)
    for friend_list in friends.values():
        for friend in friend_list:
            friend_count[friend] += 1
    return friend_count
```

计算完成后，根据好友数量对friend_count字典进行排序，找到在当前用户中，关系网最大、最密集的人。代码如下：

```
friend_count = count_friends(friends)
from operator import itemgetter
best_friends = sorted(friend_count.items(), key=itemgetter(1), reverse=Tru
```

建立一循环，凑够150个用户的好友数据后，该循环就会结束。遍历best_friends字典（按照在现有用户中的好友数多少排序），找到还没有获取好友列表的用户，然后获取他的好友列表，更新friends列表。最后，再次查找谁是最受欢迎的1：

1friend_count字典更新后，刚刚找到的好友信息已经被包括进来了。——译者注

```
while len(friends) < 150:
    for user_id, count in best_friends:
        if user_id not in friends:
```

```
            break
        friends[user_id] = get_friends(t, user_id)
    for friend in friends[user_id]:
        friend_count[friend] += 1
    best_friends = sorted(friend_count.items(),
      key=itemgetter(1), reverse=True)
```

上述代码运行结束后，就可以得到150名用户的好友数据。

> 你可能想把最大循环数设置得小一点，比如40或50（甚至可以先跳过这部分代码）。先敲完本章剩余代码，感觉下效果如何。再把循环数设置为150，等待几个小时，再来看一下结果。

鉴于上述采集数据的程序大约要运行两个小时，最好把中间结果保存下来，因为有时不得不关闭计算机。使用json库，就可以轻松把好友字典保存到文件里：

```
import json
friends_filename = os.path.join(data_folder, "python_friends.json")
with open(friends_filename, 'w') as outf:
    json.dump(friends, outf)
```

使用json.load函数，从文件中加载数据：

```
with open(friends_filename) as inf:
    friends = json.load(inf)
```

7.1.4　创建图

到现在为止，已经拿到了用户列表和他们的好友列表,很多用户是其他用户的好友。用这些存在好友关系的数据就能构建一张图（虽然这里好友关系不必是双向的）。

图由一组顶点和边组成。顶点通常表示对象——在这个例子中，顶点代表用户。边表示用户A是用户B的好友。由于顶点的顺序是有特殊含义的，该图称为有向图，因为用户A是用户B的好友并不代表用户B是用户A的好友2。可以用NetworkX库实现图关系的可视化。

> 2这里的“好友”译成“关注的人”更好理解，表示一种单向关系。其实在现实生活中，也存在这种“单相思”的情况，你把他人当成是最好的朋友，人家未必如此。——译者注

可以用pip安装NetworkX库：pip3 install networkx。

首先，使用NetworkX创建有向图。根据习惯，导入NetworkX后给它一个简称nx（虽然不是必须这么做）。代码如下：

```
import networkx as nx
G = nx.DiGraph()
```

我们只将150名核心用户彼此间的好友关系绘制成图像3，其他好友关系由于数据量很大难以可视化（成千上万个，作图有难度）。把核心用户作为顶点，添加到图中。代码如下：

3注意这里“图像”和“图”的区别，前者指的是绘画、可视化意义上点、线条等形状的组合。后者是一种抽象的数据结构。文中“作图”表示绘制图像。——译者注

```
main_users = friends.keys()
G.add_nodes_from(main_users)
```

接着要创建边。如果第二个用户是第一个用户的好友，那么就在这两个顶点之间建立一条边。遍历所有的核心用户：

```
for user_id in friends:
    for friend in friends[user_id]:
```

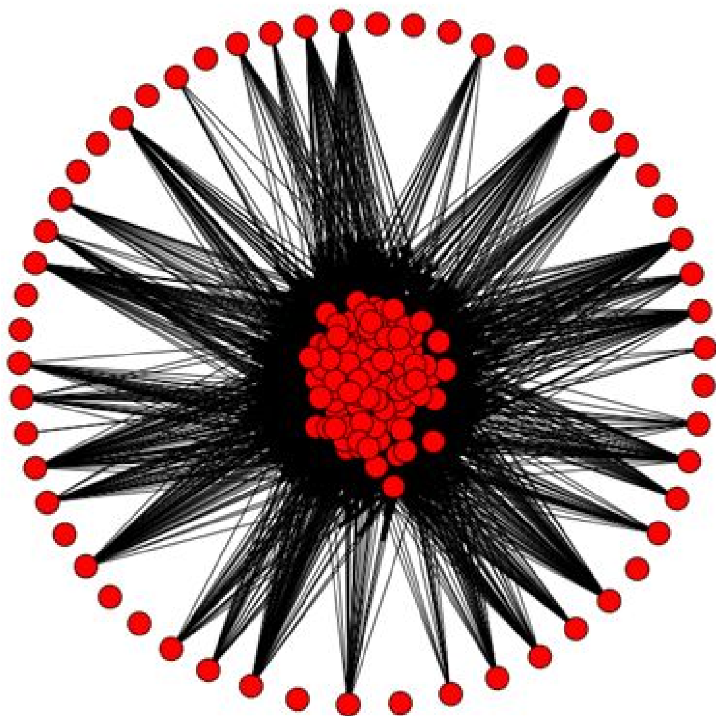
确保好友在核心用户之列（当下先不关注非核心用户），然后为两者之间添加边。代码如下：

```
        if friend in main_users:
          G.add_edge(user_id, friend)
```

用NetworkX的draw函数为创建好的图绘制图像，draw函数要用到matplotlib库。在当前笔记本文件中作图，需要使用inline函数。代码如下：

```
%matplotlib inline
nx.draw(G)
```

生成的图像理解起来有点困难；有几个顶点之间边较少，但大多数顶点之间边较多。

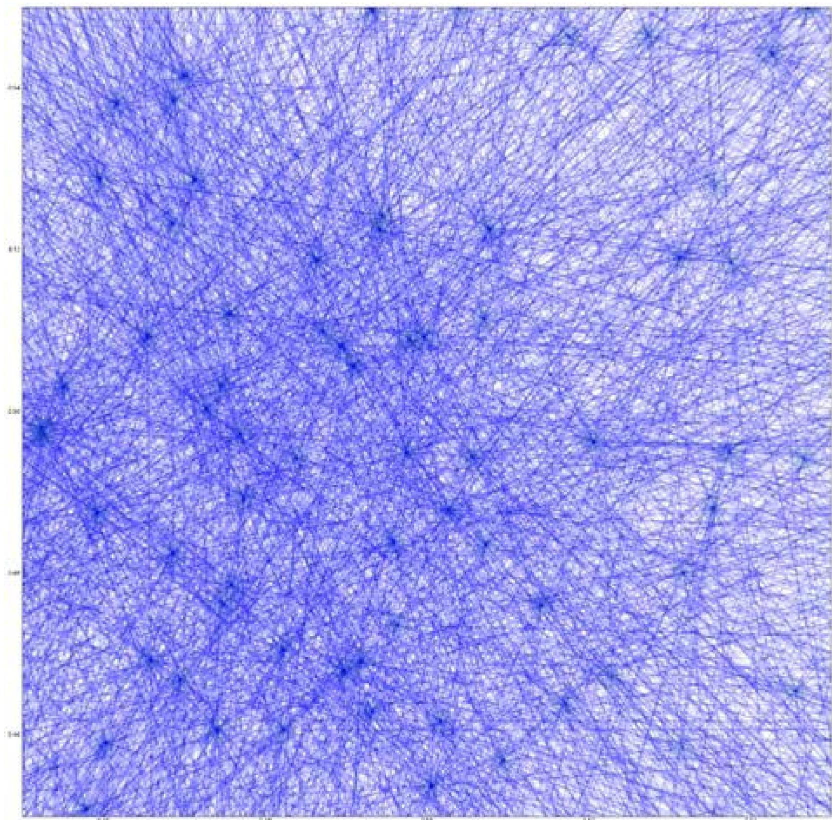


可以借助用于处理图像生成的pyplot函数设置图像大小。导入pyplot，把图像尺寸设置得大一点，调用NetworkX的draw函数（NetworkX使用pyplot函数作图）：

```
from matplotlib import pyplot as plt
plt.figure(3,figsize=(20,20))
nx.draw(G, alpha=0.1, edge_color='b')
```

生成的图像太大一页放不下，但是大图像的好处是：图的轮廓看得更清楚。在我创建的图中，一部分用户互为好友，但是其他绝大部分用户之间不存在好友关系。我将上述代码中的alpha设置得很小，这样边的颜色就为蓝色，将上图放大后从中间位置截取了下面这张图。

如你所见，中间位置各顶点之间连接有多紧密！



这与我们选择新用户的方法有关——选择那些已经在图中有着丰富连接的用户，因此他们可能只会把这个圈子扩展得更大些。一般而言，一个社交网络用户所拥有的连接数遵循幂定律。只有少数用户拥有很多连接数，大部分用户只有很少的连接数，描述这种关系的函数图像呈现出长尾形状。这里的数据集不遵从该幂定律，因为后来的核心用户产生于已有核心用户的共同好友。

7.1.5　创建用户相似度图

本章的任务是向拥有共同好友的用户推荐彼此。再次重申，逻辑是：如果两个用户有共同的好友，那么这两个用户高度相似。那么，可以向他们推荐彼此。

在现有图（边表示关注关系）基础上创建一个新图。新图中，顶点仍

然是用户，边要升级为带权重的。带权重的边指的是边有了权重属性。权重越大表明两顶点相似度越高。但是，权重与应用场景有关。如果权重代表距离，那么权重越低表示相似度越高。

对于这个应用，边的权重表示由该条边所连接的两个用户之间的相似度（以共同好友数量为基础）。新图中的边没有方向性，因为A和B之间的相似度等于B和A之间的相似度。

计算两个列表之间的相似度有多种方法。就拿我们这里的两个用户来说，可以简单统计他们好友列表中共同好友数量。然而，这样做有个问题：好友更多的用户，共同好友数量通常也更多。于是，可以再除以他们拥有的不同好友的数量实现数据的规范化，得到的正是杰卡德相似系数（Jaccard Similarity）。

杰卡德相似系数总是在0到1之间，代表两者重合的比例。正如第**2**章讲到的，规范化是数据挖掘的一个重要方法，要坚持使用（除非有充分理由不这样做）。

在计算杰卡德相似系数时，需要用两个集合（好友数据）交集的元素数量除以两个集合并集的元素数量。这里要用到集合操作，但是好友数据存储在`friends`列表中，所以首先要把列表转换为集合。代码如下：

```
friends = {user: set(friends[user]) for user in friends}
```

创建一函数**4**，计算两个好友集合之间的相似度：

> **4**注意函数体`return`语句需要缩进。——译者注

```
def compute_similarity(friends1, friends2):
return len(friends1 & friends2) / len(friends1 | friends2)
```

现在就可以创建加权图。本章后面会多次用到创建图的功能，因此将其封装成函数，留意下`threshold`参数：

```
def create_graph(followers, threshold=0):
    G = nx.Graph()
```

使用嵌套循环遍历用户数据，跳过两层循环中用户相同的情况：

```
    for user1 in friends.keys():
        for user2 in friends.keys():
            if user1 == user2:
                continue
```

计算两个用户之间边的权重：

```
                weight = compute_similarity(friends[user1],
    friends[user2])
```

只有边的权重大于阈值，才会保存这条边，以确保只保存我们认为重要的边——例如，权重为0的边就没意义。阈值默认为0，因此会添加所有的边。本章稍后再指定一个合理的阈值，先这么用着吧。代码如下：

```
            if weight >= threshold:
```

如果权重大于等于阈值，把两个用户添加到图中（如果已经存在，不要重复添加）：

```
                G.add_node(user1)
                G.add_node(user2)
```

然后为这两个用户之间添加边，把权重设置为刚计算出来的相似度：

```
                G.add_edge(user1, user2, weight=weight)
```

循环结束后，图创建完毕，返回该图：

```
    return G
```

调用上述函数就能生成一个图。由于没有设置阈值，即使权重为0的边，也被创建出来。代码如下：

```
G = create_graph(friends)
```

我们创建的图各顶点之间连接得很紧密——所有顶点之间都有边，虽然很多边的权重为0。在作图时，可以根据边的权重来指定所用线条的粗细——粗线表示权重大。

由于顶点数量较多，作图时考虑适当增加图像尺寸，顶点之间的边看上去会更加清楚：

```
plt.figure(figsize=(10,10))
```

在绘制带权重的边之前，需要先画出顶点。NetworkX根据特定标准用布局信息来决定顶点和边的位置。绘制网络图很困难，特别是顶点数量较多时，虽然有很多现成的布局方式，但是否好用很大程度上取决于你的数据集、个人喜好及作图目的。我发现spring_layout非常好用，除此之外，还有circular_layout（如果其他方式不好用，还可以试试这个）、random_layout、shell_layout和spectral_layout。

> 访问http://networkx.lanl.gov/reference/drawing.html，了解更多NetworkX布局方法。虽然用draw_graphviz选项使复杂程度有所增加，但效果很好，如果追求更好的视觉效果，值得研究一下，可考虑将其用到实际工作中。

使用spring_layout布局方法：

```
pos = nx.spring_layout(G)
```

使用pos布局方法，确定顶点位置：

```
nx.draw_networkx_nodes(G, pos)
```

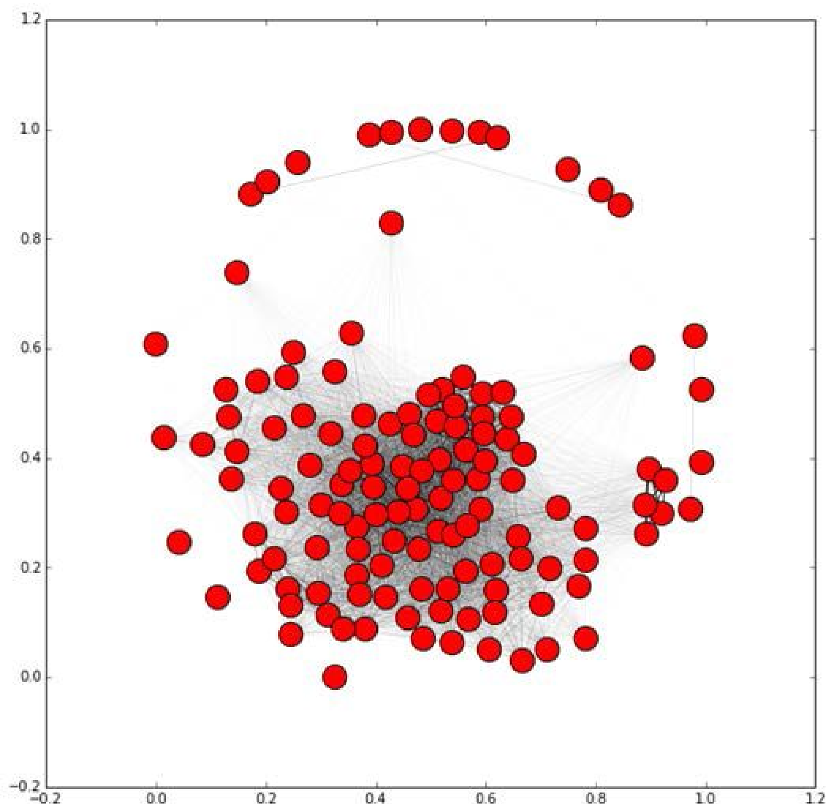
接下来，绘制边。遍历图中的每条边，获得其权重：

```
edgewidth = [ d['weight'] for (u,v,d) in G.edges(data=True)]
```

绘制各条边：

```
nx.draw_networkx_edges(G, pos, width=edgewidth)
```

最终图像具体长什么样取决于你的数据，但大体形状应该与我的一致：图中大量顶点之间有较多连接，少数顶点与网络其他顶点之间连接较少。



与本章前一个图的区别在于：该图顶点之间的边表示的是相似度而不再是好友关系（虽然这两者有相似之处）。可以从当前这个图中抽取信息用于好友推荐。

7.2　寻找子图

利用相似度函数求出每个用户与其他用户之间的相似度，根据相似度进行排序，找出与当前用户最相似的，把他推荐给当前用户——与推荐产品时的做法相同。然而，我们也可以找出批量相似度很大的用户。可以建议将这些用户组成一个群，向他们定向投放广告，或者即

使只是向他们推荐群内的其他成员做好友。

找出这些相似用户群的任务叫作聚类分析。它的复杂程度一般分类算法比不了，可以说具有一定难度。例如，评价分类结果相对比较容易——比较使用分类器得到的结果与实际结果（训练集）就能知道正确率。但是聚类分析缺乏这样一个事实标准，评价结果时，只好根据经验来看分簇结果是否合乎情理。聚类分析另一个复杂之处在于，不能用事先标注好的数据进行训练——只好在使用聚类数学模型的基础上，求得近似的分组结果，而不是按照用户所期望的那样将数据分为一个个明确的类别。

7.2.1　连通分支

一种最简单的聚类方法就是找到图中的连通分支。一个连通分支是图中由边连接在一起的一组顶点，不要求顶点之间必须两两连接。但是，连通分支的任意两个顶点之间，至少存在一条路径。

> 连通分支计算时不考虑边的权重；只检查边是否存在。因此，下面代码将删除权重过低的边。5

5因为不论权重高低，只要有边连接的顶点就会被算到连通分支里，而一个连通分支被看作是一簇具有相似特点的用户，为了保证簇内用户具有较高的相似度，需要事先把权重低的边过滤掉。——译者注

NetworkX提供用于计算连通分支的函数，在图上调用即可。首先，用前面定义的create_graph函数创建一个新图，但这次指定阈值为0.1，只保留权重至少为0.1的边。

```
G = create_graph(friends, 0.1)
```

接着使用NetworkX提供的函数寻找图中的连通分支：

```
sub_graphs = nx.connected_component_subgraphs(G)
```

为了了解连通分支到底有多大，遍历找到的连通分支，输出其基本信息：

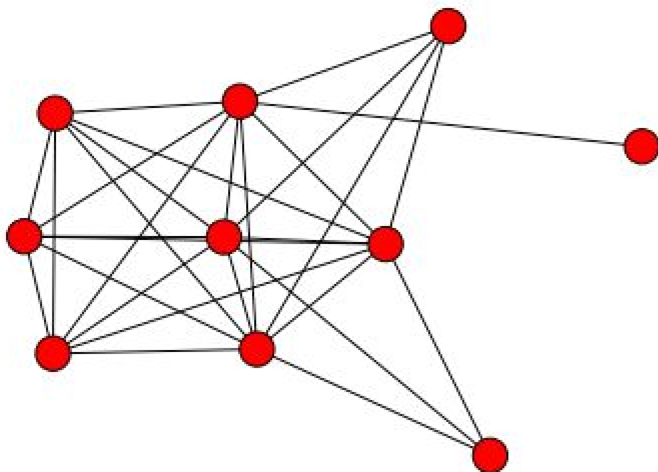
```
for i, sub_graph in enumerate(sub_graphs):
    n_nodes = len(sub_graph.nodes())
    print("Subgraph {0} has {1} nodes".format(i, n_nodes))
```

从输出结果，就能了解到每个连通分支有多大。在本例中，大的子图有62个用户，很多小子图只有十几个甚至更少的用户。

可以通过调整阈值，找出不同的连通分支。这是因为阈值越高，顶点之间的边越少，因此连通分支规模更小，数量更多。设置更高的阈值，看看效果：

```
G = create_graph(friends, 0.25)
sub_graphs = nx.connected_component_subgraphs(G)
for i, sub_graph in enumerate(sub_graphs):
    n_nodes = len(sub_graph.nodes())
    print("Subgraph {0} has {1} nodes".format(i, n_nodes))
```

上述代码找到的连通分支数量多，每个连通分支所包含的顶点数量少。之前那个最大的子图至少被拆为三个部分，每部分都不超过10个用户。下图为其中一个连通分支，边也画出来了。因为这是一个连通分支，所以它里面的顶点跟大图中其他顶点之间不存在边（阈值至少为0.25）。



可以用不同的颜色把所有连通分支都画出来。因为各连通分支之间没有连接，因此没必要把它们画到一张图中。顶点和连通分支没有确定位置，把它们画到一张图上，反而会令人产生误解，以为它们之间的

位置关系是确定的。鉴于此，我们可以把它们画到不同的图上。

在新单元格中，获得连通分支及它们的总数量。

```
sub_graphs = nx.connected_component_subgraphs(G)
n_subgraphs = nx.number_connected_components(G)
```

> 上述代码中，sub_graphs是生成器而不是连通分支列表。因此，需要用nx.number_connected_components找出连通分支的总数；由于NetworkX存储信息的方式比较特别，无法使用len函数。这正是重新计算连通分支的原因所在。

要显示所有的连通分支，图像得足够大。在用pyplot绘制图像时，让图像大小随着连通分支数量的增加而增加：

```
fig = plt.figure(figsize=(20, (n_subgraphs * 3)))
```

接下来，遍历所有的连通分支，为每一个连通分支作图。add_subplot的几个参数分别为图的行数、图的列数及图所在位置。我作图时使用了三列，你可以尝试其他值（记得同时修改行数和列数）：

```
for i, sub_graph in enumerate(sub_graphs):
    ax = fig.add_subplot(int(n_subgraphs / 3), 3, i)
```

pyplot默认显示坐标轴标签，在这里没有实际意义。因此把这个功能关掉：

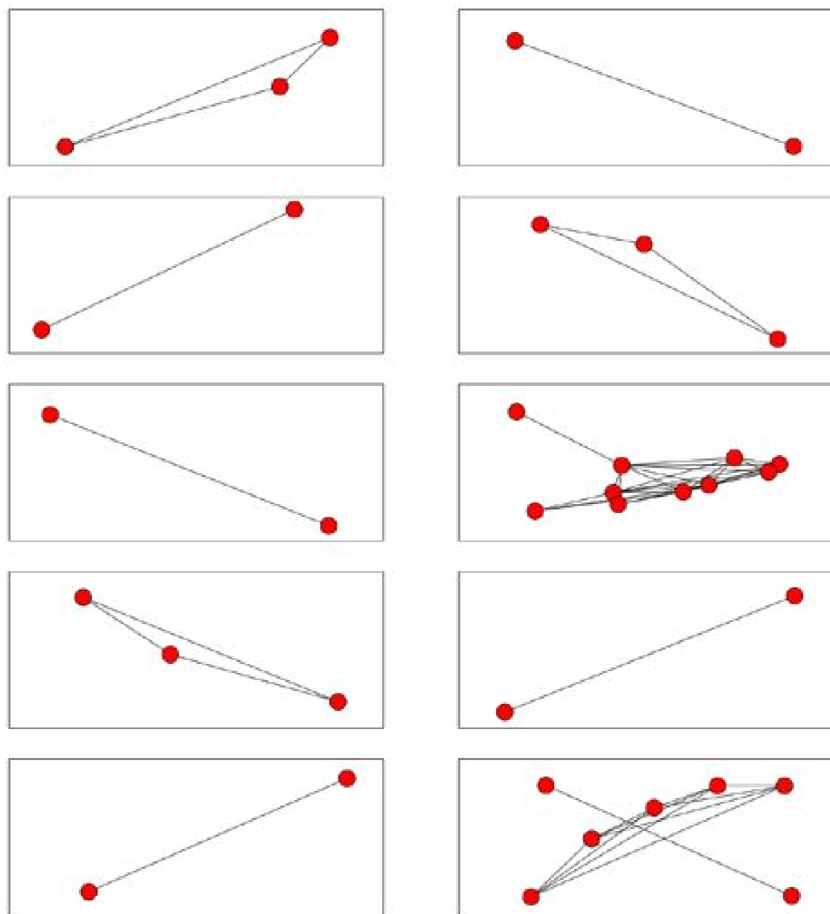
```
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

然后再绘制顶点和边（用ax参数绘制相应的子图）。绘图之前需要设置好布局：

```
    pos = nx.spring_layout(G)
    nx.draw_networkx_nodes(G, pos, sub_graph.nodes(), ax=ax,
      node_size=500)
```

```
    nx.draw_networkx_edges(G, pos, sub_graph.edges(), ax=ax)
```

从图像中就能了解到各连通分支顶点数量及连接方式。



7.2.2 优化参数选取准则

连通分支查找算法依赖于阈值参数，由其决定是否在图中添加边。换句话说，阈值控制着找到的连通分支数量和连通分支的大小。既然阈值如此关键，阈值多大才是最合适的？这个问题主观性很强，没有明确的答案。它是任何聚类分析任务都要面对的一个主要问题。

然而，我们可以事先确定什么样的结果才是令人满意的，再根据理想中的结果来寻找选取参数的准则。作为一般性规则，它应该能够使

得：

- 同一簇（连通分支）内的个体尽可能相似
- 不同簇内的个体尽可能不相似

轮廓系数（Silhouette Coefficient）就是一种对上述两点的量化方法。给定一个个体，其轮廓系数定义如下：

$$s = \frac{b-a}{\max(a,b)}$$

其中a为簇内距离，表示与簇内其他个体之间的平均距离。b为簇间距离，也就是与最近簇内各个个体之间的平均距离。

总轮廓系数为每个个体轮廓系数的均值。总轮廓系数接近最大值1时，表示每个簇内的个体相似度很高，不同簇之间距离较远。总轮廓系数接近0时，表示所有的簇重合在一起，各簇之间距离很小。总轮廓系数接近最小值-1时，表示个体出现在错误的簇内，把它们分到其他簇效果可能会更好。

利用这种选取准则，找到一种解决方案（找到合适的阈值），通过调整阈值，使总轮廓系数达到最大值。为此，创建一函数，接收阈值，计算总轮廓系数。

然后，把它传给SciPy的`optimize`模块的`minimize`函数，该函数通过调整参数值，找到函数的最小值。但我们想最大化总轮廓系数，可SciPy没有提供寻找最大值的函数。因此，对总轮廓系数取反后再找最小值（与直接求最大值是一回事）。

`scikit-learn`库提供计算轮廓系数的函数`sklearn.metrics.silhouette_score`，但是它与SciPy的`minimize`函数所要求的形式不一致。`minimize`函数要求变量参数（阈值）在其他参数的前面。这就需要把`friends`字典传进去计算图。函数声明如下：

```
def compute_silhouette(threshold, friends):
```

使用阈值参数创建图后，检查它是否至少有两个顶点：

```
    G = create_graph(friends, threshold=threshold)
```

```
    if len(G.nodes()) < 2:
```

轮廓系数的定义要求至少有两个顶点（才能计算距离）。如果少于两个顶点，认为这个问题无效，没法计算下去。有几种处理方法，最简单的是返回一个很小的值。轮廓系数最小值为-1，返回-99，表明问题无效。任何有效的值都比这个值大很多。代码如下：

```
        return -99
```

然后抽取连通分支：

```
    sub_graphs = nx.connected_component_subgraphs(G)
```

轮廓系数的定义还要求至少有两个连通分支（才能计算不同簇之间的距离），并且至少其中一个连通分支有两个顶点（计算簇内距离）。检测这些条件是否满足，如果不满足，返回无效值。代码如下：

```
    if not (2 <= nx.number_connected_components() < len(G.nodes())
      - 1):
        return -99
```

接着，需要获取标识着顶点被分到哪个连通分支的标签。遍历所有的连通分支，用字典保存顶点及其所属的连通分支。代码如下：

```
label_dict = {}
for i, sub_graph in enumerate(sub_graphs):
    for node in sub_graph.nodes():
        label_dict[node] = i
```

遍历图中所有顶点，依次获取到每个顶点的标签。需要分两步来做，先按一定顺序取到图，再遍历。因为图中顶点没有明确的顺序，但是只要没有改动图，顶点会维持现有顺序。这就表明，只要没有改动图，在图上调用`.nodes()`方法，返回的顶点顺序总是一致的。代码如下：

```
labels = np.array([label_dict[node] for node in G.nodes()])
```

注意，轮廓系数函数接收的是距离矩阵，而不是图。因此，要想办法把图转换为矩阵，同样分两步来做。首先，使用NetworkX的to_scipy_sparse_matrix函数把图转换为矩阵形式：

```
X = nx.to_scipy_sparse_matrix(G).todense()
```

写作本书时，scikit-learn实现的轮廓系数函数不支持稀疏矩阵。因此，需要调用todense()函数。很显然，这是个馊主意——通常应该用稀疏矩阵，因为数据密集的情况很罕见。这个例子这么用没问题，因为数据集相对较小；而再大点的数据集就不要这么做了。

> 建议你使用VMEASURE或调整互信息（Adjusted Mutual Information）评价稀疏数据集。scikit-learn实现了这两种方法，但是两者所用到的参数差别很大。

请注意，图中边的权重表示相似度而不是距离。对于距离而言，值越大，相似度越小。可以用可能的最大相似度（1）减去现有相似度，把相似度转化为距离：

```
    X = 1 - X
```

既然已经创建好距离矩阵和标签，现在就可以计算轮廓系数了。指定metric参数值为precomputed，否则X矩阵将被当作特征矩阵而不是距离矩阵（scikit-learn几乎到处都默认使用的特征矩阵）。函数最后返回计算得到的轮廓系数：

```
    return silhouette_score(X, labels, metric='precomputed')
```

> 有两处用到取反操作。第一处是求距离时对相似度取反，这很有必要，因为scikitlearn计算轮廓系数的函数使用距离矩阵作为参数。第二处是对轮廓系数取反，这样才能用SciPy的optimize模块。

还有个小问题没有解决。上述函数返回的轮廓系数越大越好，而SciPy的optimize模块只定义了minimize函数，它是一个损失函数，值越小越好。需要对轮廓系数取反，定义一函数把打分函数转化为损失函数。

```

```

```
def inverted_silhouette(threshold, friends):
    return -compute_silhouette(threshold, friends)
```

上述函数为用原函数创建的一个新函数。新函数调用时，为原函数传入与之前相同的参数，只不过新函数在最后返回值时进行取反操作。

现在，就可以进行优化操作了。调用minimize函数，把用于取反操作的函数inverted_silhouette传进来：

```
result = minimize(inverted_silhouette, 0.1, args=(friends,))
```

参数说明如下。

- inverted_silhouette：对我们要最小化的函数compute_silhouette进行取反操作，将其变为损失函数。
- 0.1：我们一开始猜测阈值为0.1时，函数取到最小值。
- options={'maxiter':10}：只进行10轮迭代（增加迭代次数，效果可能更好，但运行时间也会相应增加）。
- method='nelder-mead'：使用下山单纯形法（Nelder-Mead）优化方法（SciPy提供的优化方法）。
- args=(friends,)：向被优化的函数传入friends字典参数。

> 上述程序运行时间较长。创建图的函数不是很快，计算轮廓系数的函数也不快。减少maxiter的值，减少迭代次数，能缩短运行时间，但那样很可能找到的是次优阈值。

运行完上述函数，我得到的最优阈值为0.135，这时有10个连通分支。最小化函数返回的值是-0.192，然而不要忘记前面进行了取反操作，所以实际的轮廓系数是0.192。这个值为正数，表明各簇比起重合更像是分散的（好事）。可以运行其他模型，看下轮廓系数是否更大，各簇是否更分散。

可以把这个结果用到用户推荐上——如果用户属于一个连通分支，可以向他推荐该连通分支的其他用户。简要回顾下针对用户推荐的研究

历程，先是用杰卡德相似系数寻找用户间的相似度，再用连通分支把用户分成不同的簇，然后使用最优化方法找到最好的模型。

然而，大量用户可能与其他用户没有很好的联系，因此需要用不同的算法找到他们所属的簇。

7.3 小结

本章探讨了社交网络图以及如何对其进行聚类分析。还探讨了如何加载和使用在第6章创建的分类模型。

用来自Twitter的数据，创建了好友关系图，根据用户的好友，检测用户之间的相似度。我们认为共同好友更多的用户更相似，考虑到好友数量可能差别很大，对其进行了规范化处理。根据用户的相似度进行推测，是一种常用的知识（比如年龄或谈论的主题）推断方法。本章用下面的逻辑向用户推荐好友——如果他们关注用户X，用户Y和X相似，那么他们可能喜欢用户Y。这种方法与前几章从交易数据中挖掘相似商品有异曲同工之妙。

我们的目标是推荐用户，使用聚类分析方法能够找到不同的用户簇，主要步骤有根据相似度创建加权图，从图中寻找连通分支。创建图时用到了`NetworkX`库。

用轮廓系数评价聚类效果，其原则是簇内距离最小和簇间距离最大。轮廓系数越大，聚类效果越好。在寻找最合适的阈值时，用到了SciPy的`optimize`模块。

本章还比较了几对意义相反的概念。对于两者之间的相似度这个概念，值越大，表明两者之间更相像。相反，对于距离而言，值越小，两者更相像。另外一对是损失函数和打分函数。对于损失函数，值越小，效果越好（也就是损失越少）。而对于打分函数，值越大，效果越好。

下一章将介绍怎样从另外一种数据类型——图像中抽取特征。接下来，将讨论如何用神经网络识别图像中的数字，编写程序实现自动化破解验证码。

第 8 章　用神经网络破解验证码

理解图像中的信息一直是数据挖掘领域的一个难题，直到最近几年才开始得到真正解决。图像检测和理解算法已相当成熟，几大厂商使用这些算法研制的监测系统已投入商用，用来处理实际问题。这些系统能够理解和识别视频画面中的人和物体。

从图像中抽取信息很难。图像包含大量原始数据，图像的标准编码单元——像素——提供的信息量很少。图像——特别是照片——可能存在一系列问题，比如模糊不清、离目标太近、光线很暗或太亮、比例失真、残缺、扭曲等，这会增加计算机系统抽取有用信息的难度。

本章介绍如何用神经网络识别图像中的字母，从而自动识别验证码。验证码的设计初衷是便于人类理解，而不易被计算机识破。验证码的英文名叫作CAPTCHA，它取自以下短语中几个单词的首字母“Completely Automated Public Turing test to tell Computers and Humans Apart”，意思是能够区别计算机和人类的全自动的公共图灵测试。很多网站都在注册、评论系统中使用验证码，以防止别人恶意注册虚假账号或发布垃圾评论。

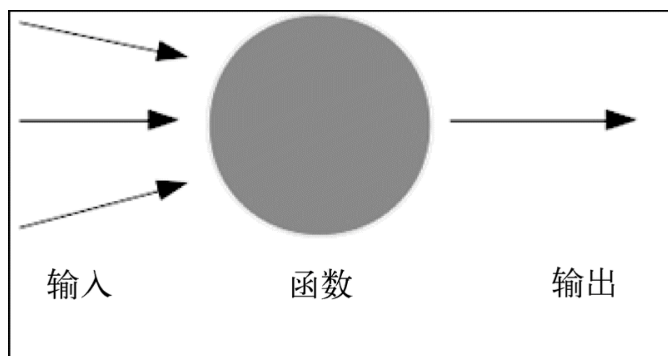
本章主要介绍如下内容。

- 神经网络
- 创建验证码和字母数据集
- 用`scikit-image`库处理图像数据
- 神经网络库`PyBrain`
- 从图像中抽取基本特征
- 使用神经网络进行更大规模的分类任务
- 用后处理提升效果

8.1　人工神经网络

神经网络算法最初是根据人类大脑的工作机制设计的。然而，该领域

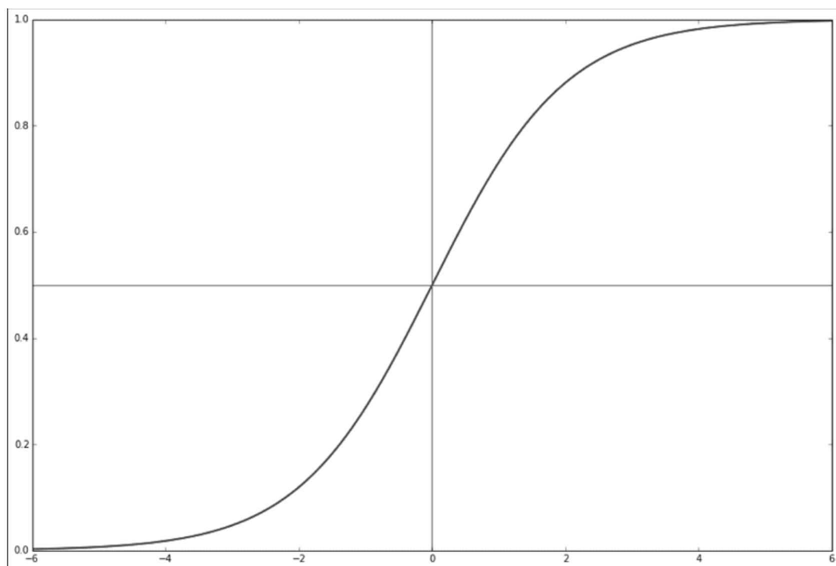
近年所取得的进展主要得益于数学而不是生物学。神经网络由一系列相互连接的神经元组成。每个神经元都是一个简单的函数，接收一定输入，给出相应输出。



神经元可以使用任何标准函数来处理数据，比如线性函数，这些函数统称为激活函数（activation function）。一般来说，神经网络学习算法要能正常工作，激活函数应当是可导（derivable）和光滑的。常用的激活函数有逻辑斯谛函数，函数表达式如下（x 为神经元的输入，k、L 通常为1，这时函数达到最大值）。

$$f(x) = \frac{L}{1 + \mathrm{e}^{-k(x-x_0)}}$$

下图为 x 取-6到6之间的函数图像。



两条红线的交点表示 x 为0时，函数值为0.5。

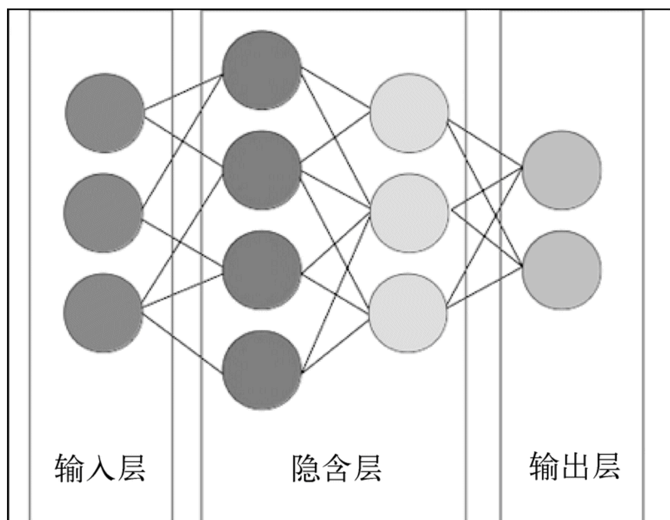
每个神经元接收几个输入，根据这几个输入1，计算输出。这样的一个个神经元连接在一起组成了神经网络，对数据挖掘应用来说，它非常强大。这些神经元紧密连接，密切配合，能够通过学习得到一个模型，使得神经网络成为机器学习领域最强大的概念之一。

> 1对带权重的输入加总。——译者注

神经网络简介

用于数据挖掘应用的神经网络，神经元按照层级进行排列。第一层，也就是输入层，接收来自数据集的输入。第一层中的每个神经元对输入进行计算，把得到的结果传给第二层的神经元。这种叫作前向神经网络。本章暂且把它简称为神经网络。此外，还有几种神经网络，适用于不同的应用。第11章会讲到另外一种类型的神经网络。

神经网络中，上一层的输出作为下一层的输入，直到到达最后一层：输出层。输出结果表示的是神经网络分类器给出的分类结果。输入层和输出层之间的所有层被称为隐含层，因为在这些层中，其数据表现方式，常人难以理解。大多数神经网络至少有三层，而如今大多数应用所使用的神经网络层次比这多得多。



我们优先考虑使用全连接层，即上一层中每个神经元的输出都输入到下一层的所有神经元。实际构建神经网络时，就会发现，训练过程中，很多权重都会被设置为0，有效地减少边的数量。比起其他连接模式，全连接神经网络更简单，计算起来更快捷。

神经元激活函数通常使用逻辑斯谛函数，每层神经元之间为全连接，创建和训练神经网络还需要用到其他几个参数。创建过程，指定神经网络的规模需要用到两个参数：神经网络共有多少层，隐含层每层有多少个神经元（输入层和输出层神经元数量通常由数据集来定）。

训练过程还会用到一个参数：神经元之间边的权重。一个神经元连接到另外一个神经元，两者之间的边具有一定的权重，在计算输出时，用边的权重乘以信号的大小（signal，第一个神经元的输出）。如果边的权重为0.8，神经元激活后，输出值为1，那么下一个神经元从前面这个神经元得到的输入就是0.8。如果第一个神经元没有激活，值为0，那么输出到第二个神经元的值就是0。

神经网络大小合适，且权重经过充分训练，它的分类效果才能精确。大小合适并不是越大越好，因为神经网络过大，训练时间会很长，更容易出现过拟合训练集的情况。

> 通常，开始时使用随机选取的权重，训练过程中再逐步更新。

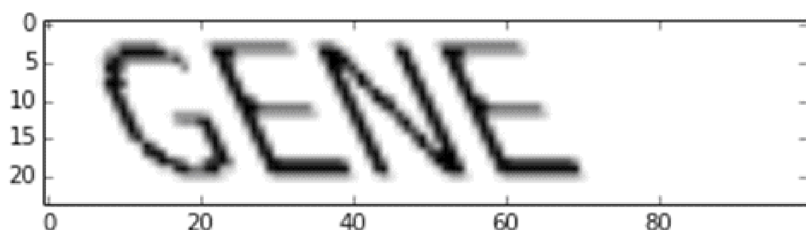
设置好第一个参数（网络的大小），再从训练集中训练得到边的权重

参数后，就能构造分类器。然后，就可以用它进行分类，跟前几章用到的分类器算法很像。但是，首先需要准备训练集和测试集。

8.2 创建数据集

本章，我们将扮演骇客这个角色，我们想编写破解网站验证码的程序，这样我们就有能力在别人网站上发布恶意广告。需要注意的是，我们要破解的验证码，难度比不上现在网上常用的；还有就是发表垃圾广告不太道德。

我们只使用长度为4个字母的英文单词作为验证码，如下图所示。



我们的目标是编写程序还原图像中的单词，步骤如下。

(1) 把大图像分成只包含一个字母的4张小图像。

(2) 为每个字母分类。

(3) 把字母重新组合为单词。

(4) 用词典修正单词识别错误。

我们的验证码破解算法做出了以下几个假设。首先，验证码中的单词是一个完整的、有效的英文单词，其长度为4个字母（实际上，生成和破解验证码，我们都使用同一个词典）。其次，单词全部字母均为大写形式，不使用符号、数字或空格。为了增加难度，在生成图像时对单词使用不同的错切（shear）变化效果。

8.2.1 绘制验证码

接下来，我们编写创建验证码的函数，目标是绘制一张含有单词的图像，对单词使用错切变化效果。绘制图像要用到`PIL`库，错切变化需要使用`scikit-image`库。`scikit-image`库能够接收`PIL`库导出的

numpy数组格式的图像数据，因此这两个工具可以结合使用。



PIL和scikitimage库都可以用pip安装：

```
pip install PIL
pip install scikit-image
```

首先，导入即将用到的库和模块。导入numpy，从PIL中导入Image等绘图函数。

```
import numpy as np
from PIL import Image, ImageDraw, ImageFont
from skimage import transform as tf
```

接着，创建用于生成验证码的基础函数。这个函数接收一个单词和错切值（通常在0到0.5之间），返回用numpy数组格式表示的图像。该函数还提供指定图像大小的参数，因为后面还会用它生成只包含单个字母的测试数据。代码如下：

```
def create_captcha(text, shear=0, size=(100, 24)):
```

我们使用字母L来生成一张黑白图像，为ImageDraw类初始化一个实例。这样，我们就可以用PIL绘图。代码如下：

```
    im = Image.new("L", size, "black")
    draw = ImageDraw.Draw(im)
```

指定验证码文字所使用的字体。这里要用到字体文件，下面代码中的文件名（Coval.otf）应该指向文件存放位置（我把它放到当前笔记本所在目录）。

```
    font = ImageFont.truetype(r"Coval.otf", 22)
    draw.text((2, 2), text, fill=1, font=font)
```



你可以从开源字库（Open Font Library）下载所

需字体文件Coval.otf：http://openfontlibrary.org/en/font/bretan。

把PIL图像转换为numpy数组，以便用scikit-image库为图像添加错切变化效果。scikit-image大部分计算都使用numpy数组格式。代码如下：

```
    image = np.array(im)
```

然后，应用错切变化效果，返回图像。

```
    affine_tf = tf.AffineTransform(shear=shear)
    image = tf.warp(image, affine_tf)
    return image / image.max()
```

上面最后一行代码对图像特征进行归一化处理，确保特征值落在0到1之间。归一化处理可在数据预处理、分类或其他阶段进行。

现在，我们可以轻松生成图像，使用pyplot绘制图像。首先设定魔术方法%matplotlib，指定在当前笔记本作图，导入pyplot。代码如下：

```
%matplotlib inline
from matplotlib import pyplot as plt
```

生成验证码图像并显示它。

```
image = create_captcha("GENE", shear=0.5)
plt.imshow(image, cmap='Greys')
```

生成的图像就是本节开头你见到的那张，效果还不错吧。

8.2.2　将图像切分为单个的字母

虽然我们的验证码是单词，但是我们不打算构造能够识别成千上万个单词的分类器，而是把大问题转化为更小的问题：识别字母。

验证码识别算法的下一步是分割单词，找到其中的字母。具体做法是，创建一个函数，寻找图像中连续的黑色像素，抽取它们作为新的

小图像。这些小图像（或者至少应该）就是我们要找的字母。

首先，导入图像分割函数要用到的label、regionprops函数。

```
from skimage.measure import label, regionprops
```

图像分割函数接收图像，返回小图像列表，每张小图像为单词的一个字母，函数声明如下：

```
def segment_image(image):
```

我们要做的第一件事就是检测每个字母的位置，这就要用到scikit-image的label函数，它能找出图像中像素值相同且又连接在一起的像素块。这有点像第7章中的连通分支。

label函数的参数为图像数组，返回跟输入同型的数组。在返回的数组中，图像连接在一起的区域用不同的值来表示，在这些区域以外的像素用0来表示。代码如下：

```
    labeled_image = label(image > 0)
```

抽取每一张小图像，将它们保存到一个列表中。

```
    subimages = []
```

scikit-image库还提供抽取连续区域的函数：regionprops。遍历这些区域，分别对它们进行处理。

```
    for region in regionprops(labeled_image):
```

这样，通过region对象就能获取到当前区域的相关信息。我们这里要用到当前区域的起始和结束位置的坐标。

```
        start_x, start_y, end_x, end_y = region.bbox
```

用这两组坐标作为索引就能抽取到小图像（image对象为numpy数

组，可以直接用索引值），然后，把它保存到subimages列表中。代码如下：

```
        subimages.append(image[start_x:end_x,start_y:end_y])
```

最后（循环外面），返回找到的小图像，每张（希望如此）小图像包含单词的一个字母区域。没有找到小图像的情况，直接把原图像作为子图返回。代码如下：

```
    if len(subimages) == 0:
        return [image,]
    return subimages
```

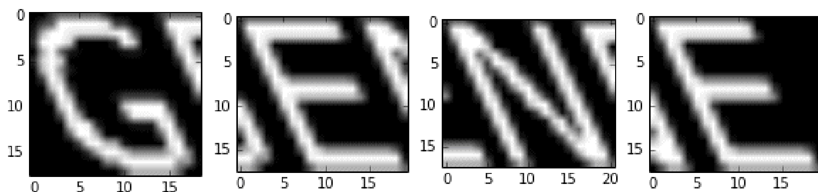
使用刚定义的这个函数，就能从前面生成的验证码中找到小图像。

```
subimages = segment_image(image)
```

还可以像下面这样查看每张小图像。

```
f, axes = plt.subplots(1, len(subimages), figsize=(10, 3))
for i in range(len(subimages)):
    axes[i].imshow(subimages[i], cmap="gray")
```

结果如下。



图像切割效果还不错，但是你可能注意到，每张小图像都多少带有相邻字母的一部分。

8.2.3　创建训练集

使用图像切割函数就能创建字母数据集，其中字母使用了不同的错切效果。然后，就可以训练神经网络分类器来识别图像中的字母。

首先，指定随机状态值，创建字母列表，指定错切值。这里几乎没有新内容，numpy的arange函数你可能没用过，它跟Python的range函数类似——只不过arange函数可以和numpy的数组一起用，步长可以使用浮点数。代码如下：

```
from sklearn.utils import check_random_state
random_state = check_random_state(14)
letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
shear_values = np.arange(0, 0.5, 0.05)
```

再来创建一个函数（用来生成一条训练数据），从我们提供的选项中随机选取字母和错切值。代码如下：

```
def generate_sample(random_state=None):
    random_state = check_random_state(random_state)
    letter = random_state.choice(letters)
    shear = random_state.choice(shear_values)
```

返回字母图像及表示图像中字母属于哪个类别的数值。字母A为类别0，B为类别1，C为类别2，以此类推。代码如下：

```
    return create_captcha(letter, shear=shear, size=(20, 20)),
      letters.index(letter)
```

在上述函数体的外面，调用该函数，生成一条训练数据，用pyplot显示图像。

```
image, target = generate_sample(random_state)
plt.imshow(image, cmap="Greys")
print("The target for this image is: {0}".format(target))
```

调用几千次该函数，就能生成足够的训练数据。把这些数据传入到numpy的数组里，因为数组操作起来比列表更容易。代码如下：

```
dataset, targets = zip(*(generate_sample(random_state) for i in
range(3000)))
dataset = np.array(dataset, dtype='float')
targets =  np.array(targets)
```

我们共有26个类别，每个类别（字母）用从0到25之间的一个整数表示。神经网络一般不支持一个神经元输出多个值，但是多个神经元就能有多个输出，每个输出值在0到1之间。因此，我们对类别使用一位有效码编码方法，这样，每条数据就能得到26个输出。如果结果像某字母，使用近似于1的值；如果不像，就用近似于0的值。代码如下：

```
from sklearn.preprocessing import OneHotEncoder
onehot = OneHotEncoder()
y = onehot.fit_transform(targets.reshape(targets.shape[0],1))
```

我们用的库不支持稀疏矩阵，因此，需要将稀疏矩阵转换为密集矩阵。代码如下：

```
y = y.todense()
```

8.2.4　根据抽取方法调整训练数据集

得到的数据集跟即将使用的方法有较大出入。数据集每条数据都是一个恰好为20像素见方的字母。我们所使用的方法是从单词中抽取字母，而这可能会挤压图像，使图像偏离中心或者引入其他问题。

理想情况下，训练分类器所使用的数据应该与分类器即将处理的数据尽可能相似。但实际中，经常有所不同，但是应尽量缩小两者之间的差别。

对于验证码识别实验，最理想的情况是，从实际的验证码中抽取字母，并对它们进行标注。为了节省时间，我们只在训练集上运行分割函数，返回分割后得到的字母图像 。

我们需要用到scikit-image库中的resize函数，因为我们得到的小图像并不总是20像素见方。代码如下：

```
from skimage.transform import resize
```

现在，就可以对每条数据运行segment_image函数，将得到的小图像调整为20像素见方。代码如下：

```
dataset = np.array([resize(segment_image(sample)[0], (20, 20)) for
sample in dataset])
```

```
```

最后，创建我们的数据集。dataset数组为三维的，因为它里面存储的是二维图像信息。由于分类器接收的是二维数组，因此，需要将最后两维扁平化。

```
X = dataset.reshape((dataset.shape[0], dataset.shape[1] * dataset.
shape[2]))
```

使用scikit-learn中的train_test_split函数，把数据集切分为训练集和测试集。代码如下：

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, train_size=0.9)
```

8.3 训练和分类

接下来，我们就来构造神经网络分类器，接收图像，预测图像中的（单个）字母是什么。

我们将使用之前创建的单个字母训练集。数据集本身很简单。每张图像为20像素大小，每个像素用1（黑）或0（白）来表示。每张图像有400个特征，将把它们作为神经网络的输入。输出结果为26个0到1之间的值。值越大，表示图像中的字母为该值所对应的字母（输出的第一个值对应字母A，第二个对应字母B，以此类推）的可能性越大。

我们用PyBrain库来构建神经网络分类器。

> 跟我们之前见到的所有库一样，PyBrain也可以用pip来安装：pip install pybrain。

PyBrain库使用自己的数据集格式，好在创建这种格式的训练集和测试集并不太难。代码如下：

```
from pybrain.datasets import SupervisedDataSet
```

首先，遍历我们的训练集，把每条数据添加到一个新的SupervisedDataSet实例中。代码如下：

```
training = SupervisedDataSet(X.shape[1], y.shape[1])
for i in range(X_train.shape[0]):
    training.addSample(X_train[i], y_train[i])
```

然后，遍历测试集，同样把每条数据添加到一个新的SupervisedDataSet实例中。代码如下：

```
testing = SupervisedDataSet(X.shape[1], y.shape[1])
for i in range(X_test.shape[0]):
    testing.addSample(X_test[i], y_test[i])
```

现在就可以创建神经网络了。我们将创建一个最基础的、具有三层结构的神经网络，它由输入层、输出层和一层隐含层组成。输入层和输出层的神经元数量是固定的。数据集有400个特征，那么第一层就需要有400个神经元，而26个可能的类别表明我们需要26个用于输出的神经元。

确定隐含层的神经元数量可能相当困难。如果神经元数量过多，神经网络呈现出稀疏的特点，这时要训练足够多的神经元恰当地表示数据就有困难，这往往会导致过拟合训练数据的问题。相反，如果神经元过少，每个对分类结果的贡献过大，再加上训练不充分，就很可能产生低拟合现象。我发现一开始用漏斗形状不错，即隐含层神经元数量介于输入和输出层之间。本章，隐含层用100个神经元，你可以尝试其他值，看看能不能取得更好的效果。

导入buildNetwork函数，指定维度，创建神经网络。第一个参数X.shape[1]为输入层神经元的数量，也就是特征数（数据集X的列数）。第二个参数指隐含层的神经元数量，这里设置为100。第三个参数为输出层神经元数量，由类别数组y的形状来确定。最后，除去输出层外，我们每层使用一个一直处于激活状态的偏置神经元（bias neuron，它与下一层神经元之间有边连接，边的权重经过训练得到）。代码如下：

```
from pybrain.tools.shortcuts import buildNetwork
net = buildNetwork(X.shape[1], 100, y.shape[1], bias=True)
```

现在，我们就可以训练神经网络，寻找合适的权重值。但是如何训练神经网络呢？

8.3.1 反向传播算法

反向传播算法（back propagation，backprop）的工作机制为对预测错误的神经元施以惩罚。从输出层开始，向上层层查找预测错误的神经元，微调这些神经元输入值的权重，以达到修复输出错误的目的。

神经元之所以给出错误的预测，原因在于它前面为其提供输入的神经元，更确切来说是由这两个神经元之间边的权重及输入值决定的。我们可以尝试对权重进行微调。每次调整的幅度取决于以下两个方面：神经元各边权重的误差函数的偏导数和一个叫作学习速率的参数（通常使用很小的值）。计算出函数误差的梯度，再乘以学习速率，用总权重减去得到的值。下面将给出例子。梯度的符号由误差决定，每次对权重的修正都是朝着给出正确的预测值努力。有时候，修正结果为局部最优（local optima），比起其他权重组合要好，但所得到的各权重还不是最优组合。

反向传播算法从输出层开始，层层向上回溯到输入层。到达输入层后，所有边的权重更新完毕。

`PyBrain`提供了backprop算法的一种实现，在神经网络上调用`trainer`类即可。代码如下：

```
from pybrain.supervised.trainers import BackpropTrainer
trainer = BackpropTrainer(net, training, learningrate=0.01,
weightdecay=0.01)
```

backprop算法在训练集上进行迭代，每次都对权重进行微调。当误差减小的幅度很小时，表明算法无法进一步缩小误差，继续训练已无意义，这时就可以停止算法运行。理论上，等误差不再缩小时，再停止运行。这个过程叫作算法收敛。但实际上我们不会等到误差停止缩小，因为这通常要花很长时间且效果有限。

此外，更简单的做法是，运行代码固定的步数（epoch）。每一步所包含的次数越多，所用时间也就越多，效果越好（每一步的提升效果呈下降趋势）。我们这里训练时，使用了20步，数量再多些，效果可能会有所改善（也许幅度很小）。代码如下：

```
trainer.trainEpochs(epochs=20)
```

运行前面的代码，这可能要花几分钟时间，长短取决于硬件，然后我

们就能在测试集上进行预测。PyBrain提供了相应函数，只要在trainer实例上调用即可。

```
predictions = trainer.testOnClassData(dataset=testing)
```

得到预测值后，可以用scikit-learn计算F1值。

```
from sklearn.metrics import f1_score
print("F-score: {0:.2f}".format(f1_score(predictions,
                                         y_test.argmax(axis=1) )))
```

F1值为0.97，对于相对较小的模型来说，这个结果很了不起。别忘了我们的特征值只是简单的像素值，神经网络竟然知道怎么使用它们。

既然构建好了具有高精度的字母分类器，接下来看下怎么用它来识别单词，实现验证码破解功能。

8.3.2　预测单词

我们想分别识别每张小图像中的字母，然后把它们拼成单词，完成验证码识别。

我们来定义一个函数，接收验证码，用神经网络进行训练，返回单词预测结果。

```
def predict_captcha(captcha_image, neural_network):
```

使用前面定义的图像切割函数segment_image抽取小图像。

```
    subimages = segment_image(captcha_image)
```

最终的预测结果——单词，将由小图像中的字母组成。这些小图像根据位置进行排序，从而保证拼接后得到的单词中各字母处在正确的位置上。

```
    predicted_word = ""
```

遍历四张小图像。

```
    for subimage in subimages:
```

每张小图像不太可能正好是20像素见方。调整大小后，才能用神经网络处理。

```
        subimage = resize(subimage, (20, 20))
```

把小图像数据传入神经网络的输入层，激活神经网络。这些数据将在神经网络中进行传播，返回输出结果。神经网络在前面已经创建好了，现在只需激活它。代码如下：

```
        outputs = net.activate(subimage.flatten())
```

神经网络输出26个值，每个值都有索引号，分别对应letters列表中有着相同索引的字母，每个值的大小表示与对应字母的相似度。为了获得实际的预测值，我们取到最大值的索引，再通过letters列表找到对应的字母。例如，如果第五个值最大，那么预测结果就为字母E。代码如下：

```
        prediction = np.argmax(outputs)
```

把上面得到的字母添加到正在预测的单词中。

```
        predicted_word += letters[prediction]
```

循环结束后，我们就已经找到了单词的各个字母，将其拼接成单词，最后返回这个单词。

```
    return predicted_word
```

可以使用下面代码来做下测试。尝试不同的单词，看看可能会遇到什么错误，别忘了我们的神经网络只能处理大写字母。

```
word = "GENE"
```

```
captcha = create_captcha(word, shear=0.2)
print(predict_captcha(captcha, net))
```

我们可以把上面代码整合到一个函数里，便于进行多次尝试。这里沿用之前每个单词只包含四个字母的假设，降低预测任务的难度。删除prediction = prediction[:4]，试试看可能会出什么错误。代码如下：

```
def test_prediction(word, net, shear=0.2):
    captcha = create_captcha(word, shear=shear)
    prediction = predict_captcha(captcha, net)
    prediction = prediction[:4]
    return word == prediction, word, prediction
```

上述函数返回结果依次为预测结果是否正确、验证码中的单词和预测结果的前四个字符。

上面代码能正确识别单词GENE，但是其他单词会出错。正确率如何？我们借助NLTK模块创建单词数据集，只使用长度为4的单词。从NLTK中导入words，代码如下：

```
from nltk.corpus import words
```

words实际上为corpus（语料库）对象，调用它的words()方法才能取到里面的单词。下面代码还加了长度为4的过滤条件。代码如下：

```
valid_words = [word.upper() for word in words.words() if len(word) ==
4]
```

识别得到的所有长度为4的单词，分别统计识别正确和错误的数量。

```
num_correct = 0
num_incorrect = 0
for word in valid_words:
    correct, word, prediction = test_prediction(word, net,
                                                shear=0.2)
if correct:
        num_correct += 1
    else:
        num_incorrect += 1
```

```
print("Number correct is {0}".format(num_correct))
print("Number incorrect is {0}".format(num_incorrect))
```

正确识别2832个单词，错误识别2681个，正确率仅为51%。而字母识别率为97%，两者差距太大。到底怎么回事？

首先，来看下正确率本身。其余条件相同的情况下，我们有四个字母，每个字母的正确率为97%，四个字母都正确的话，正确率约为88%（约为0.974）。一个字母出错将导致整个单词识别错误。

其次，错切值对正确率有影响。这次创建数据集时，随机从0到0.5之间选取一个数作为错切值。先前测试时错切值为0.2。错切值为0时，正确率为75%；错切值取0.5时，正确率只有2.5%。错切值越大，正确率越低。

另外一个原因在于我们之前随机选取字母组成单词，而字母在单词中的分布不是随机的。例如，字母E显然就比Q等其他字母使用频率更高。使用频度较高，但却常常被识别错误的字母，也会导致错误率上升。

我们可以把经常识别错误的字母统计出来，用二维混淆矩阵来表示。每行和每列均为一个类别（字母）。

矩阵的每一项表示一个类别（行对应的类）被错误识别为另一个类别（列对应的类）的次数。例如，`(4,2)`这一项的值为6，表示字母D有6次被错误地识别为字母B。

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(np.argmax(y_test, axis=1), predictions)
```

理想情况下，混淆矩阵应该只有处于对角线的项`(i,i)`值不为零，其余各项的值均为零，否则就是分类有误。

用`pyplot`做成图，哪些字母容易混淆就一目了然，代码如下：

```
plt.figure(figsize=(10, 10))
plt.imshow(cm)
```

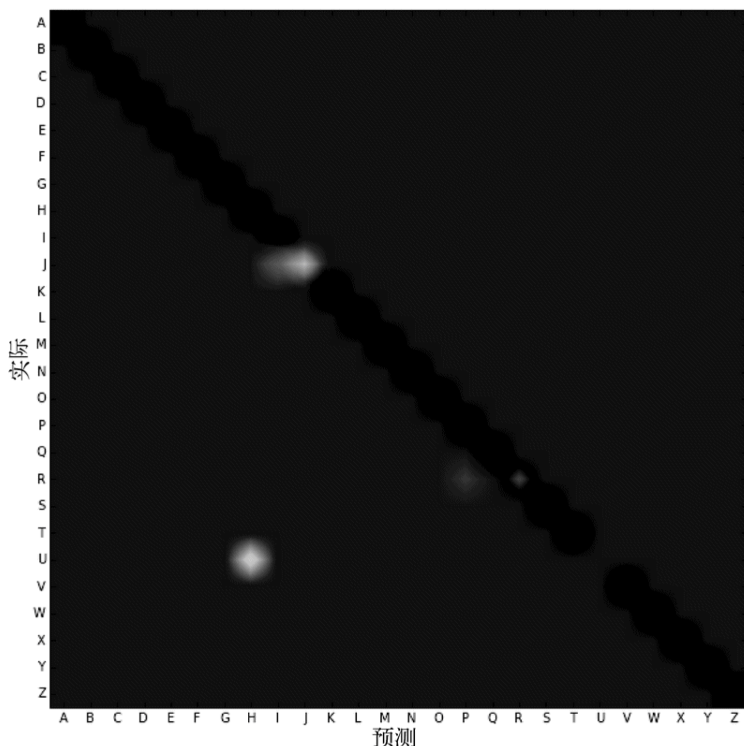
下面代码中的`tick_marks`表示刻度，在坐标轴上标出每个字母，便于理解。

```
tick_marks = np.arange(len(letters))
plt.xticks(tick_marks, letters)
plt.yticks(tick_marks, letters)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

结果如下图所示。从图中，我们就能清楚地看到主要的错误在于几乎无一例外地把U识别为H！

我们的词表中17%的单词含有字母U，这些单词几乎都会被识别错误。U的出现频率要高于H（11%的单词），我们不禁想到了一个提升正确率的简单方法（可能不够健壮）：把所有预测结果为H的，都改为U。

下节将采用更为巧妙的方法，从词典中搜索跟预测结果相似的单词。



8.4　用词典提升正确率

我们刚刚是直接返回预测结果，其实返回之前可以先检查一下词典里是否包含该词条。如果单词在词典里，那么就返回预测结果，如果不在，我们找到和预测结果相似的单词，再把它作为更新过的预测结果返回。注意，该方法是基于所有验证码中的单词都是英语单词的假设，因此，对于随机选取字母组成的验证码来说不适用。其实这正是很多验证码不使用单词的原因之一。

还有一个问题要解决——如何找到最相近的单词？方法有很多。例如，比较词长。词长差不多的两个单词更相似。然而，通常认为，在相同位置有相同字母的两个单词更相似，这就要用到编辑距离。

8.4.1　寻找最相似的单词

列文斯坦编辑距离（Levenshtein edit distance）是一种通过比较两个短字符串，确定它们相似度的方法。它不太适合扩展，字符串很长时通常不用这种方法。编辑距离需要计算从一个单词变为另一个单词所需要的步骤数。以下操作都算一步。

(1) 在单词的任意位置插入一个新字母。

(2) 从单词中删除任意一个字母。

(3) 把一个字母替换为另外一个字母。

把一个单词转变为另外一个单词所需步数越少，这两个单词越相似，反之，差别越大。

`NLTK`实现了编辑距离算法，可以直接拿来用。导入后，指定两个单词，看下它们的编辑距离。

```
from nltk.metrics import edit_distance
steps = edit_distance("STEP", "STOP")
print("The number of steps needed is: {0}".format(steps))
```

尝试不同的单词，你会发现编辑距离返回的结果跟我们的直觉很相近。编辑距离在检查拼写、听写错误和名称匹配（比如Marc和Mark的拼写形式就极易被搞混）方面用处很大。

然而，我们这里没这么复杂，只需要比较两个单词同等位置上的字母是否一致即可，不涉及字母的移动。因此，我们自己来编写距离度量函数，也就是统计同等位置上字母不同的数量。代码如下：

```
def compute_distance(prediction, word):
    return len(prediction) - sum(prediction[i] == word[i] for i in
range(len(prediction)))
```

用词长（4）减去同等位置上相同的字母的数量，得到的值越小表示两个词相似度越高。

8.4.2 组装起来

我们现在就来测试改进的预测函数，代码跟之前相似。首先，定义预测函数，这次把单词列表也传进去。

```
from operator import itemgetter
def improved_prediction(word, net, dictionary, shear=0.2):
    captcha = create_captcha(word, shear=shear)
    prediction = predict_captcha(captcha, net)
    prediction = prediction[:4]
```

到目前为止，代码跟之前的预测函数一致，还是取到预测结果的前四个字母。然而，这次还要检测单词是否在词典里。如果在，把单词作为预测结果返回；如果不在，找到最相似的单词返回。代码如下：

```
    if prediction not in dictionary:
```

计算预测结果和词典中每个单词的距离，按照距离由小到大进行排序。代码如下：

```
        distances = sorted([(word, compute_distance(prediction, word))
                            for word in dictionary],
key=itemgetter(1))
```

找到最为匹配的单词——也就是距离最小的单词——随后把这个单词作为预测结果返回。

```
        best_word = distances[0]
        prediction = best_word[0]
```

函数返回结果跟之前相同：预测结果是否正确，验证码中的单词和预测结果。

```
    return word == prediction, word, prediction
```

测试代码中需要变动的地方用粗体表示。

```
num_correct = 0
num_incorrect = 0
for word in valid_words:
    correct, word, prediction = improved_prediction(word, net, valid_words
        if correct:
            num_correct += 1
        else:
            num_incorrect += 1
print("Number correct is {0}".format(num_correct))
print("Number incorrect is {0}".format(num_incorrect))
```

上述代码运行时间稍长（计算预测结果跟词典中每个单词之间的距离要耗费一定时间），最终正确识别3037个单词，错误识别2476个，正确率较之前有4个百分点的提升，达到55%。由于预测结果跟词典中很多单词的相似度一致，算法随机从一组最相似的单词中选择最佳的，所以算法效果提升很有限。例如，词典中第一个单词AANI（埃及神话中狗头猿猴），跟其他44个单词距离相同。选中的几率就只有1/44。

如果我们作弊，预测结果只要从最相似的一组词中随机挑一个就算是正确的话，那么预测结果正确率将高达78%（代码请见本书配套的代码包）。

要进一步提升正确率，得改变距离测量方法，看能不能利用之前得到的混淆矩阵，找到经常被混淆的字母或其他有用信息。逐步改进效果，是很多数据挖掘算法的一个特点，这跟科学实验方法是一脉相承的——产生想法，做实验，分析结果，再根据结果做进一步的改进。

8.5　小结

本章的任务是根据验证码图像的像素值识别里面的单词。当然，为了简化难度，验证码使用由四个字母组成的英语单词。网站上实际使用的验证码要比这复杂得多——也应该这样！但是，只要对上面所讲的算法进行改进，就能通过神经网络，使用跟这里类似的方法破解更为复杂的验证码。`scikit-image`库提供大量图像处理工具，比如从图像中抽取不同形状的小图像和改善图像对比度的方法等。这些方法有

助于破解验证码。

我们把识别单词的难题转化为识别字母的小问题，创建前向神经网络识别图像中的字母，取得了97%的准确率。

神经网络由一组组神经元连接而成，神经元为基本的计算单元，每个神经元都只包含一个函数。然而，所有的神经单元连接在一起就能解决复杂程度很高的问题。神经网络是深度学习的基础，后者可是当今数据挖掘领域最受关注的方向之一。

尽管识别字母正确率很高，但是单词识别正确率陡然降至50%多点。原因有很多，其实这恰好说明把算法从实验环境搬到真实环境会遇到各种各样的困难。

使用词典，寻找最匹配的单词，能够提升正确率。我们考虑使用常用的编辑距离表示单词间的相似度；但是我们只关注字母错误而不深究与哪些编辑步骤（插入、删除）相关，因此简化了距离计算方法。最终正确率有所提升，但是还有很多要改进的地方。

下一章将继续研究字符串比较，尝试找出文档的主人（从多名可能的作者中）——只根据文档内容而不使用其他信息！

第 9 章　作者归属问题

文本挖掘任务作者分析（authorship analysis）的目标是只根据作品内容找出作者独有的特点，比如年龄、性别或背景。作者归属（authorship attribution）是作者分析的一个细分领域，研究目标是从一组可能的作者中找到文档真正的主人。这是一种典型的分类任务。作者分析任务一般采用标准的数据挖掘方法，比如交叉检验、特征抽取和分类算法等。

本章将把前几章学到的数据挖掘方法整合起来，解决作者归属问题，从而掌握数据挖掘整个过程。我们界定问题，讨论相关背景和知识，抽取特征，创建流水线实现分类。我们将比较两类特征的效果：功能词和N元语法。最后，深入分析结果。数据集会用到两种，一开始使用图书，然后增加难度，使用噪音较多的真实电子邮件语料。

本章主要介绍如下内容。

- 特征工程和如何根据应用选择特征
- 带着新问题，重新回顾词袋模型
- 特征类型和字符N元语法模型
- 支持向量机
- 数据集清洗

9.1　为作品找作者

作者分析可以从文体学（stylometry）中找到它的一席之地，文体学分析研究的是作者的写作风格，所依据的理念是每个人在语言掌握上有微小差异，这会反映到写作中，通过分析作品之间的细微差别，就可以把作者区分开来。

过去一直靠人工统计、分析来解决作者分析问题，而这些工作恰好是计算机所擅长的，这表明可以借助数据挖掘技术来实现自动化。如今作者分析的研究工作几乎都是用数据挖掘方法来解决，但仍有不少研究偏重于人工分析作品语言风格。

作者分析问题衍生出很多更细的问题，主要有以下几个。

- 作者画像：根据作品界定作者的年龄、性别或其他特性。例如，通过观察一个人讲英语的方式，判断英语是否为他的母语。
- 作者验证：根据作者已有作品，推断其他作品是否也是他写的。拿法庭断案场景来讲更好理解，例如，分析嫌犯的写作风格（内容方面），以确定勒索信是不是他写的。
- 作者聚类：作者验证问题的延伸，用聚类分析方法把作品按照作者进行分类。

然而，作者分析研究领域最常见的还是作者归属问题，即如何从一群作者中找到作品的真正主人。

9.1.1　相关应用和使用场景

作者分析有很多应用场景，比如证实作者是谁，证实几本书有相同的作者，或者寻找社会媒体账号的主人。

从历史意义上来说，作者分析可用来核实某些文档是否真由公认的作者所写。有些作品的作者就存在争议，比如莎士比亚的几部戏剧，美国建国时期的联邦党人文集等。

单凭作者分析无法确定作者到底是谁，但是能够提供支持或反驳某一观点的依据。例如，在检验一首十四行诗是否是莎士比亚所写前，我们可以先分析他的若干部剧作，弄清楚他的写作风格。

作者分析问题最近几年也被用来确定社会媒体账号到底是谁在使用。例如，一个恶意用户可能在不同社交平台创建多个账号。把它们关联起来后，便于监管部门跟踪幕后的黑手——例如，当扰乱其他用户时。

举个陈旧点的例子，法庭上视作者分析为核心技术，主要靠它来提供文档是否出自嫌犯之手的证据。例如，嫌疑犯可能因为发骚扰邮件而被起诉。作者分析就能确定邮件是否是嫌犯所写。另一个在法庭中的使用场景是，解决作品归属纠纷。比如，遇到两个人就一本书到底是谁写的而纠缠不已的情况，法庭就可以借助作者分析技术找到作者可能是谁的证据。

作者分析也并不是绝对可靠的。最近研究发现，即使没有经过专业训练的人，让他们隐藏自己的写作风格，也会让作者归属问题变得困难无比。研究人员还探讨了模仿别人写作风格的问题，结果发现人们模仿得很像，作者分析的结果是这些伪作出自被模仿人之手。

尽管存在很多问题，作者分析在越来越多的领域被证实十分有用，它是一个非常有意思的、值得研究的数据挖掘问题。

9.1.2 作者归属

作者归属可以看作是一种分类问题，已知一部分作者，数据集为多个作者的作品（训练集），目标是确定一组作者不详的作品（测试集）是谁写的。如果作者恰好是已知的作者里面的，这种问题叫作封闭问题。



如果作者可能不在里面，这种问题就叫作开放问题。不只是作者归属问题可以这样区分——任何数据挖掘应用，只要实际的类别可能不在训练集中的都叫作开放问题，对于这类问题进行挖掘，最终目标分两种情况，如果在训练集中，就返回找到的类别，如果不在，要给出不属于任何现有类别的提示。



在进行作者归属研究时，一般会受到两个限制。一是只能使用作品的

内容，而不能使用写作时间、印刷形式、笔迹等信息。当然也有这样的做法，把这些信息整合到模型中，但一般来说这就不是作者归属而更像是数据融合问题。

第二个限制是不考虑作品的主题；相反，关注单词用法、标点和其他文本特征。原因在于，一个人如果是多面手，可以写多个主题的内容，作品的主题就不能反映作者的实际写作风格。关注主题关键字可能会导致过度拟合训练集——模型可能只是用同一个作者的同一个主题的作品进行训练。例如，根据我写的这本书为我的写作风格建模，可能会得出“数据挖掘”这个词能代表我的风格这样的结论，而事实上，我还写一些其他主题的内容。

接下来，用于作者归属问题的流水线跟第6章所用的很相似。首先，从文本中抽取特征；然后，对抽到的特征做进一步甄选；最后，训练算法分类器，预测文档的类别（作者）。

当然有些步骤跟第6章还是存在一些差别，主要集中在特征选取方面，后面会详细讲。我们还是先来划定待解决问题的范围。

9.1.3 获取数据

本章所用的图书数据集来自于古腾堡计划网站（www.gutenberg.org），该网站收集了大量版权失效的公版文学作品。实验用到以下作家的若干作品。

- 塔金顿（Booth Tarkington，22篇）
- 狄更斯（Charles Dickens，44篇）
- 伊迪斯·内斯比特（Edith Nesbit，10篇）
- 阿瑟·柯南·道尔（Arthur Conan Doyle，51篇）
- 马克·吐温（Mark Twain，29篇）
- 理查德·弗朗西斯·伯顿爵士（Sir Richard Francis Burton，11篇）
- 埃米尔·加博里奥（Emile Gaboriau，10篇）

以上总共是7位作家的177篇作品，文本数量还是相当可观的。作品名称列表、下载链接以及自动下载图书的代码请见本书配套的代码包。

用requests库下载作品文件到数据文件夹所在的目录。首先，在Data目录下创建文件夹存放这些作品。注意，下面代码指定的文件目录跟你实际创建的文件夹目录保持一致。

```
import os
import sys
data_folder = os.path.join(os.path.expanduser("~"), "Data", "books")
```

接着，就可以使用代码包中我写好的代码，从古腾堡计划的网站下载图书。下载完后将其保存到我们上面创建的目录中。

从代码包Chapter 9文件夹中找到getdata.py，保存到笔记本文件所在目录下，在新格子中输入以下代码。

```
!load getdata.py
```

在笔记本中，按下Shift＋Enter运行该格子的代码。代码将会加载到格子中。然后再次点击代码，按下Shift＋Enter运行加载的代码。代码要运行一段时间，运行完毕后会提示你。

大体翻看一下下载的作品，你会发现大部分都包含很多噪音——从数据分析的角度看至少是这样的。每篇作品前都有大段的声明文字，开始数据分析之前，得先把它们删掉。

我们可以直接从文件里将这些免责声明删除，但是丢失数据怎么办？我们可能会丢失先前做过的改动，从而导致实验结果无法重现，因此，不如在加载文件时跳过这部分——这样再次用这些文件做实验时，也能得到同样的结果（只要原文件没变）。代码如下：

```
def clean_book(document):
```

每篇作品前后各有一行文字，标识作品的开头和结尾，作品前后为古腾堡项目的说明。既然我们能根据这两行找到作品内容，因此，就按行来切分。

```
    lines = document.split("\n")
```

遍历文档的每一行，寻找作品的开头和结尾，中间部分就是作品内

容。代码如下：

```
    start = 0
    end = len(lines)
    for i in range(len(lines)):
        line = lines[i]
        if line.startswith("*** START OF THIS PROJECT GUTENBERG"):
            start = i + 1
        elif line.startswith("*** END OF THIS PROJECT GUTENBERG"):
            end = i - 1
```

函数最后，用换行符把所有行再连接起来，得到作品内容。

```
    return "\n".join(lines[start:end])
```

现在，我们可以创建一个函数，加载所有图书，进行上述预处理操作，返回书的内容以及作家序号1。先来导入numpy。

> 1用索引来表示，详见下文。——译者注

```
import numpy as np
```

声明加载图书的函数，参数为图书所在目录books，该目录下是一系列以作者名字命名的子文件夹，图书文件就在这些子文件夹中。代码如下：

```
def load_books_data(folder=data_folder):
```

创建两个列表，分别用来存储文档和作者。

```
    documents = []
    authors = []
```

获取到books目录下所有的子文件夹。代码如下：

```
    subfolders = [subfolder for subfolder in os.listdir(folder)
                  if os.path.isdir(os.path.join(folder,
                     subfolder))]
```

遍历这些子文件夹，使用enumerate函数为这些子文件夹指定索引。

```
    for author_number, subfolder in enumerate(subfolders):
```

获取到子文件夹的绝对路径，查找里面的所有图书文件。

```
        full_subfolder_path = os.path.join(folder, subfolder)
        for document_name in os.listdir(full_subfolder_path):
```

对于找到的每一个图书文件，打开后读取里面的内容，对内容进行预处理后，将其添加到documents列表中。代码如下：

```
            with open(os.path.join(full_subfolder_path,
              document_name)) as inf:
                documents.append(clean_book(inf.read()))
```

把分配给作家的索引号添加到authors列表中，其实authors就是类别列表。

```
                authors.append(author_number)
```

函数最后返回文档和类别（把类别列表转换为numpy数组）。

```
    return documents, np.array(authors, dtype='int')
```

调用上面定义的加载图书函数，就能获得图书文档及其它们的类别。

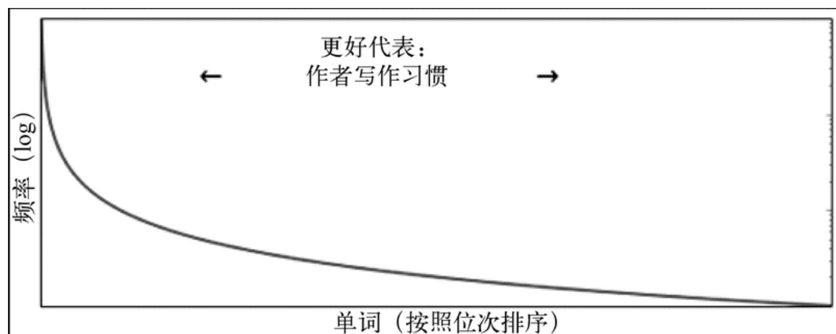
```
documents, classes = load_books_data(data_folder)
```

把这些数据加载到内存没有问题，因此，我们可以立即加载。如果数据集太大，内存装不下，较好的方法是每次只从一篇（或几篇）文档中抽取特征，把这些特征保存到文件或位于内存的矩阵中。

9.2　功能词

功能词是作者分析问题最早使用的一类特征，到现在来看，在词袋模型中使用功能词做特征效果依然不错。功能词指的是本身具有很少含义，却是组成（英语）句子必不可少的成分。例如，单词this和which的意思不是由它们本身的含义所决定，而是由它们在句子中充当的角色来决定。与功能词相对的是实词，比如tiger就有明确的含义，当人们在句子中看到这个单词时，脑海中就会浮现出大型猫科动物老虎的样子。

有些功能词的用法并不总是清晰、明确的。因此，从经验来看，选用使用频率较高的功能词做特征比较好（在所有文档中使用频率高，而不仅是在单个作者的作品中）。通常而言，使用越频繁的单词，对于作者分析能提供更多有价值的信息。相反，使用频率较低的单词，更适合用来做基于内容的文本挖掘，例如下章要讲的文档主题划分。



功能词的使用通常不是由文档内容而是由作者的使用习惯所决定，因此可以用它们来区分不同的作者。例如，很多美国人比较在意区分that和which在句子中的用法，澳大利亚等国家的人就不太在意。在使用这两个词的地方，有些澳大利亚人要么几乎无一例外地都用that，要么是which用得多一点。这样的不同点，再加上成千上万个其他的微小差别，就形成了用于作者分析的模型。

9.2.1　统计功能词

我们可以使用第6章所用到的`CountVectorizer`统计功能词。我们把包含所有要查找的单词的词汇表（vocabulary）传递进去，如果没有传词汇表（第6章就没有），它会从数据集中学习。所有单词都在训练集中（取决于其他参数）。

首先，创建功能词词汇表，用列表存储。至于确切来说哪些是功能词，哪些不是，有待商榷。我从已发表的研究成果中找到下面这些功

能词，它们还是比较可靠的。

```
function_words = ["a", "able", "aboard", "about", "above", "absent",
"according" , "accordingly", "across", "after", "against",
"ahead", "albeit", "all", "along", "alongside", "although",
"am", "amid", "amidst", "among", "amongst", "amount", "an",
"and", "another", "anti", "any", "anybody", "anyone",
"anything", "are", "around", "as", "aside", "astraddle",
"astride", "at", "away", "bar", "barring", "be", "because",
"been", "before", "behind", "being", "below", "beneath",
"beside", "besides", "better", "between", "beyond", "bit",
"both", "but", "by", "can", "certain", "circa", "close",
"concerning", "consequently", "considering", "could",
"couple", "dare", "deal", "despite", "down", "due", "during",
"each", "eight", "eighth", "either", "enough", "every",
"everybody", "everyone", "everything", "except", "excepting",
"excluding", "failing", "few", "fewer", "fifth", "first",
"five", "following", "for", "four", "fourth", "from", "front",
"given", "good", "great", "had", "half", "have", "he",
"heaps", "hence", "her", "hers", "herself", "him", "himself",
"his", "however", "i", "if", "in", "including", "inside",
"instead", "into", "is", "it", "its", "itself", "keeping",
"lack", "less", "like", "little", "loads", "lots", "majority",
"many", "masses", "may", "me", "might", "mine", "minority",
"minus", "more", "most", "much", "must", "my", "myself",
"near", "need", "neither", "nevertheless", "next", "nine",
"ninth", "no", "nobody", "none", "nor", "nothing",
"notwithstanding", "number", "numbers", "of", "off", "on",
"once", "one", "onto", "opposite", "or", "other", "ought",
"our", "ours", "ourselves", "out", "outside", "over", "part",
"past", "pending", "per", "pertaining", "place", "plenty",
"plethora", "plus", "quantities", "quantity", "quarter",
"regarding", "remainder", "respecting", "rest", "round",
"save", "saving", "second", "seven", "seventh", "several",
"shall", "she", "should", "similar", "since", "six", "sixth",
"so", "some", "somebody", "someone", "something", "spite",
"such", "ten", "tenth", "than", "thanks", "that", "the",
"their", "theirs", "them", "themselves", "then", "thence",
"therefore", "these", "they", "third", "this", "those",
"though", "three", "through", "throughout", "thru", "thus",
"till", "time", "to", "tons", "top", "toward", "towards",
"two", "under", "underneath", "unless", "unlike", "until",
"unto", "up", "upon", "us", "used", "various", "versus",
"via", "view", "wanting", "was", "we", "were", "what",
"whatever", "when", "whenever", "where", "whereas",
"wherever", "whether", "which", "whichever", "while",
"whilst", "who", "whoever", "whole", "whom", "whomever",
"whose", "will", "with", "within", "without", "would", "yet",
"you", "your", "yours", "yourself", "yourselves"]
```

既然有了功能词列表，我们就来创建功能词统计工具。后面，会把它

加到流水线中。

```
from sklearn.feature_extraction.text import CountVectorizer
extractor = CountVectorizer(vocabulary=function_words)
```

9.2.2 用功能词进行分类

接下来，导入所需的几个类，唯一的新内容支持向量机在下节会讲（现在把它看作是标准的分类算法即可）。导入用支持向量机算法进行分类的SVC类，以及其他一些我们用过的标准工作流工具。

```
from sklearn.svm import SVC
from sklearn.cross_validation import cross_val_score
from sklearn.pipeline import Pipeline
from sklearn import grid_search
```

支持向量机接收一系列参数。现阶段照我设置的参数来就行，下节再深入探讨参数值选择。我们用字典结构来组织参数。参数kernel使用linear和rbf。C的值取1或10（参数说明请见下节）。接着用网络搜索法寻找最优参数值。

```
parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
svr = SVC()
grid = grid_search.GridSearchCV(svr, parameters)
```

高斯内核（例如rbf）只适用于数据集相对较小的情况，比如特征数少于10 000。

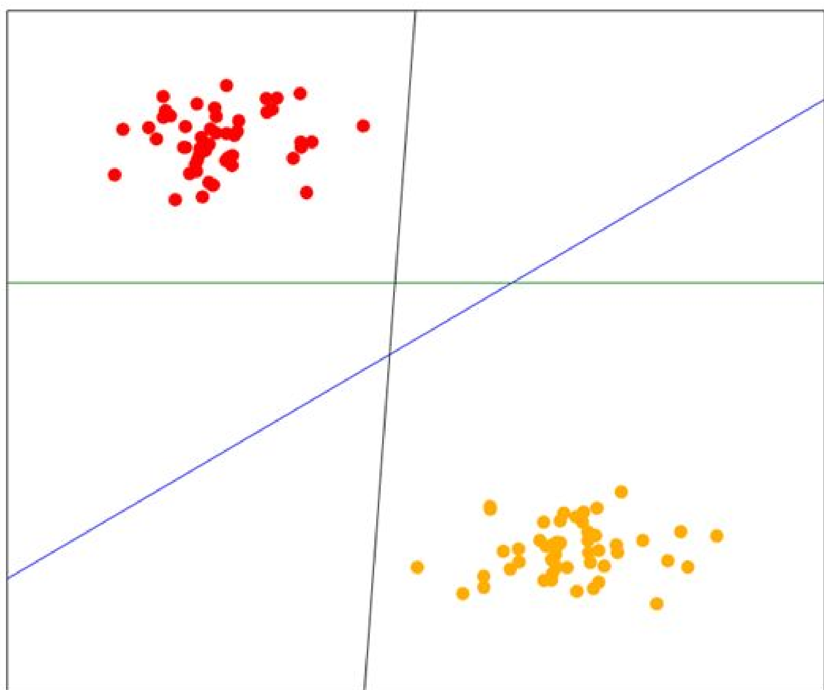
接着，创建流水线，把特征抽取和参数搜索两个步骤加入到流水线中，特征（仅功能词）抽取使用CountVectorizer类，参数搜索使用SVM。代码如下：

```
pipeline1 = Pipeline([('feature_extraction', extractor),
                      ('clf', grid)
                      ])
```

然后，使用cross_val_score对该流水线的结果进行交叉检验，正确率为0.811，大约80%的预测结果正确。对于只有7个作者来说，这个结果很好！

9.3　支持向量机

支持向量机（SVM）分类算法背后的思想很简单，它是一种二类分类器（扩展后可用来对多个类别进行分类）。假如我们有两个类别的数据，而这两个类别恰好能被一条线分开，线上所有点为一类，线下所有点属于另一类。SVM要做的就是找到这条线，用它来做预测，跟线性回归原理很像。只是SVM要找出最佳的分割线。下图中有三条线，分别为蓝色、黑色和绿色线，都能把两类数据区分开来。你说哪条分类效果更好呢？



凭直觉，人们往往会选择从左下到右上的这条线2，它把两类数据更好地分开了。也就是说，数据集中的每个点到这条线的距离都是最远的。

> 2如果你查看原书PDF文件的话，这条线指的是图中的蓝线。PDF文件获取方法请见前言部分。建议读者下载查看。彩色图毕竟比黑白图形象。——译者注

找到这样的一条线其实是最优化问题，也就是要让各点到分割线之间的距离最大化。

虽然最优化问题的公式推导不在本书讲述范围之列，我还是建议感兴趣的读者自行查看http://en.wikibooks.org/wiki/Support_Vector_Machines，了解详细步骤。此外，你还可以访问http://docs.opencv.org/doc/tutorials/ml/introduction_to_svm/introduction_to_svm.html，了解支持向量机相关内容。

9.3.1　用SVM分类

模型训练完毕，就能找到使得两类之间间隔最大的一条线。新数据点分类问题就简化为：数据点在线上还是线下？如果在线上，就被分到线上那一类；如果位于线下，自然属于线下那一类。

对于多种类别的分类问题，我们创建多个SVM分类器——每个还是二类分类器。连接多个分类器有不少方法，从中任选一种即可。最简单的方法是为每个类别创建一对多（one-versus-all）分类器，把训练数据分为两个类别——属于特定类别的数据和其他所有类别数据。对新数据进行分类时，从这些类别中找出最匹配的。大多数SVM多类分类器都能自动完成上述过程。

前面代码中有两个参数：`C`和`kernel`。`kernel`参数下节讲，`C`参数对于训练SVM来说很重要，我们现在来讲下。`C`参数与分类器正确分类比例相关，但可能带来过拟合的风险。`C`值越高，间隔越小，表示要尽可能把所有数据正确分类。`C`值越小，间隔越大——有些数据将无法正确分类。`C`值低，过拟合训练数据的可能性就低，但是分类效果可能会相对较差。

SVM（基础形式）局限性之一就是只能用来对线性可分的数据进行分类。如果线性不可分呢？这时我们就要用到内核函数。

9.3.2　内核

如果数据线性不可分，就需要将其置入更高维的空间中，加入更多伪特征直到数据线性可分（如果你加入足够多恰当的特征，总能把数据分开）。

寻找最佳分隔线时往往需要计算个体之间的内积3。对于使用点积（dot product）的函数，我们可以创建新特征而无需实际定义这些特征。因为我们不知道这些特征到底是什么样子，所以用点积很方便。我们把内核函数定义为数据集中两个个体函数的点积，而不是使用个

体自身（及构造的特征）。

> 3inner product，也称作点积。——译者注

接下来就可以计算点积（或近似值），然后就能使用它。

常用的内核函数有几种。线性内核最简单，它无外乎两个个体的特征向量的点积、带权重的特征和偏置项。多项式内核提高点积的阶数（比如2）。此外，还有高斯内核（rbf）、Sigmoind内核。前面例子中，我们比较了线性内核和`rbf`内核的效果。

这些内核能够有效地确定两类数据之间的距离，SVM可以据此对新数据进行分类。理论上讲，任何距离度量方法都可以用，但是在训练SVM时，最优化问题的复杂程度会有所不同。

`scikit-learn`实现的SVM，可以通过`kernel`参数指定内核函数，正如我们在前面例子中见过的。

9.4　字符N元语法

我们前面研究的是如何用功能词做特征预测文档的作者。接下来看下字符N元语法特征。N元语法由一系列的N个为一组的对象组成。N为每组对象的个数（对于文本来说，N通常取2到6之间的值）。基于单词的N元语法被广泛用在通常与文档主题相关的各项研究中。然而，基于字符的N元语法被证明在作者归属问题上效果很好。

我们可以把文档看成是由一系列字符组成的，从里面抽取N元语法，训练得到模型。常用模型有几个，其中标准模型跟我们之前用到的词袋模型很相似。

分别为训练语料中各个不同的N元语法创建一个特征。例如，由字母e、空格和字母t组成的N元语法`<e t>`（尖括号标识N元语法的边界，不是N元语法的内容）。然后，我们使用N元语法在训练集中的频率训练模型，再用生成的特征矩阵训练分类器。

> 基于字符的N元语法有几种不同的定义方法。比如，有些应用只选取单词内的字符，忽略空格和标点。但有些就使用这些信息（比如我们这一章）。

关于基于字符的N元语法为什么效果比较好，广为接受的理论是，人

们在写作时，往往选取那些他们讲起来很容易的单词，而字符N元模型（至少N取 2到6之间时）跟这些音素具有很好的相似关系——音素指的是我们讲话时，组成单词发音的那些声音。从这种意义上讲，用字符N元模型可以很好地模拟讲话时经常用到的单词，而这些单词形成了写作风格。这是创建新特征的通用模式：先找到哪些概念影响最终结果（写作风格）的理论依据，然后创建跟这些概念相近或是能够量化这些概念的特征。

字符N元语法矩阵的一个主要特点是数据稀疏，随着N值的增加，稀疏程度逐渐加大，且速度很快。N取2时，我们的特征矩阵中大约75%的项都是0，N取5时，93%以上的项为0。比起基于词语的N元语法矩阵来说，稀疏程度要低一些，因此，适用于单词分类的分类器处理数据，不会有太大问题。

抽取字符N元语法

我们接下来用`CountVectorizer`类来抽取N元语法，需要设置`analyzer`参数，指定N的值。

`scikit-learn`的N元语法抽取工具提供了`range`参数，允许用户同时抽取不同长度的N元语法。我们这里用不到，只抽取相同长度即可，`range`参数的两个值使用相同值即可。要抽取长度为3的N元语法，`range`的值需要设置为(3, 3)。

我们可以重用前面网格搜索代码，但是需要在新流水线中指定新的特征抽取器。

```
pipeline = Pipeline([('feature_extraction', CountVectorizer(analyzer='
char', ngram_range=(3, 3))),
                     ('classifier', grid)
                     ])
scores = cross_val_score(pipeline, documents, classes, scoring='f1')
print("Score: {:.3f}".format(np.mean(scores)))
```

功能词和字符N元模型之间存在大量隐式重合，因为字符序列更可能出现在功能词中。然而，实际上两者差别很大，字符N元模型能够捕获标点的使用特点，功能词自然做不到。例如，字符N元语法能够包含句号，而基于功能词的方法就只能使用句号前的单词。

9.5　使用安然公司数据集

安然公司（Enron）是20世纪90年代末期世界上最大的能源公司之一，年收入高达1000亿美元。2000年时，拥有20 000余名员工，看不出公司有什么严重问题。

2001年，安然丑闻发生，调查人员发现安然为谋取暴利，在全公司范围内账务存在系统性造假现象。丑闻被揭发后，安然公司的股价从2000年的90多美元一下子跌到了2001年的1美元。安然随即申请破产保护，留下的残局，五年多以后才收拾干净。

作为对安然公司调查的一部分，美国联邦能源署公开了60多万封电子邮件。从那时起，这些数据就被广泛应用于社会媒体分析、欺诈分析等众多问题的研究。这些数据用于作者归属问题研究也不错，因为我们能获得发件人及他们写的邮件，语料规模比之前见过的很多数据集都要大得多。

9.5.1　获取安然数据集

安然公司的邮件可以从卡内基梅隆大学的网站下载：https://www.cs.cmu.edu/~./enron/。

> 整个数据集为423MB，压缩格式为gzip。如果你用的不是Linux系统，可以使用免费的7zip（(http://www.7-zip.org/）等软件来解压。

下载电子邮件语料，将其解压到Data目录下。解压后的目录默认为enron_mail_20110402。

因为要做作者归属分析，我们只需要那些明确知道发件人是谁的邮件。因此，查看每位用户的发件夹——那里面存放的是他们所发的邮件。

在笔记本中，指定安然数据集所在的位置。

```
enron_data_folder = os.path.join(os.path.expanduser("~"), "Data",
  "enron_mail_20110402", "maildir")
```

9.5.2　创建数据集加载工具

我们现在就来创建一个函数，它接收几个发件人作为参数，返回他们所发送的邮件。我们需要的有效信息是邮件内容而不是邮件本身。因此，还需要邮件解析器。代码如下：

```
from email.parser import Parser
p = Parser()
```

我们后面会用到该解析器从邮件中抽取邮件内容。

因为是随机选取收件人，为了重现实验结果，我们设置随机状态。

```
from sklearn.utils import check_random_state
```

数据加载函数提供几个参数，大部分是为了确保得到的数据集类别分布相对比较均衡。有些用户发了成千上万封邮件，而有的用户只发了几十封。我们用min_docs_author参数指定每个发件人至少发过10封邮件，用max_docs_author参数指定最多从一个用户那里抽取100封邮件。我们还用num_authors限定了收件人数量——默认为10。代码如下：

```
def get_enron_corpus(num_authors=10, data_folder=data_folder,
                     min_docs_author=10, max_docs_author=100,
                     random_state=None):
    random_state = check_random_state(random_state)
```

接下来，获取到安然公司员工的邮箱，邮箱其实就是data_folder文件夹中各子文件夹的名称。随机对得到的邮箱列表进行排序。只要随机状态相同，实验结果就可以重现。

```
    email_addresses = sorted(os.listdir(data_folder))
    random_state.shuffle(email_addresses)
```

你可能很好奇我们既然后面接着要随机调整顺序，那为什么还要事先进行排序。因为os.listdir函数每次返回结果不一定相同，在使用该函数前先排序，从而保持返回结果的一致性。然后我们用随机状态的shuffle函数随机选取一组收件人。如果需要的话，随机状态函数可以返回跟之前相同的结果。

然后，我们创建文档列表、类别列表，author_num指的是每个新发件人的类别编号。这次我们不用enumerate函数，因为有些发件人我们用不到。例如，发过不够10封邮件的，就不会用。代码如下：

```
documents = []
classes = []
author_num = 0
```

我们还需要记录我们所用到的收件人及他们的编号。数据挖掘过程用不到，但是在可视化时能用到，便于我们确定收件人。authors字典用于将用户名和类别关联起来。代码如下：

```
authors = {}
```

接下来，遍历邮箱文件夹，查找它下面名字中含有“sent”的表示发件箱的子文件夹。代码如下：

```
for user in email_addresses:
  users_email_folder = os.path.join(data_folder, user)
  mail_folders = [os.path.join(users_email_folder,
    subfolder) for subfolder in os.listdir(users_email_folder)
                if "sent" in subfolder]
```

获得子文件夹中的每一封邮件。我们用到了try-except语句，因为有些用户的发件箱目录还有子目录。要获取目录层次更深的邮件，我们的代码还需要做些改进，现在我们先跳过这些用户。代码如下：

```
    try:
      authored_emails = [open(os.path.join(mail_folder,
        email_filename), encoding='cp1252').read()
      for mail_folder in mail_folders
      for email_filename in os.listdir(mail_folder)]
    except IsADirectoryError:
        continue
```

接下来，检测是否获取到了至少10封邮件（或min_docs_author指定的其他值）。

```
    if len(authored_emails) < min_docs_author:
        continue
```

下一步，如果发件人发了大量邮件，只取前100封（具体数量由max_docs_author指定）。

```
        if len(authored_emails) > max_docs_author:
            authored_emails = authored_emails[:max_docs_author]
```

解析邮件，获取邮件内容。我们对邮件头部不感兴趣——发件人所能改动的内容比较少，因此对作者分析没有多大用处。然后把邮件内容添加到数据集中。

```
        contents = [p.parsestr(email)._payload for email in
          authored_emails]
        documents.extend(contents)
```

把该发件人添加到类别列表中，每一封邮件添加一次。

```
        classes.extend([author_num] * len(authored_emails))
```

记录该收件人的编号，再把编号加1，以便下一个收件人使用。

```
        authors[user] = author_num
        author_num += 1
```

检测收件人数量是否达到我们设置的值，如果是，跳出循环，返回数据集。

```
        if author_num >= num_authors or author_num >=
          len(email_addresses):
            break
```

返回邮件数据集和类别以及收件人字典。

```
    return documents, np.array(classes), authors
```

在上述函数的外面，调用这个函数，我们就能得到数据集。我们使用随机状态14（全书都是），但是你可以尝试其他值或者将其设置为none，这样每次调用这个函数时将随机获取一组数据。

```
documents, classes, authors = get_enron_corpus(data_folder=enron_data_
folder, random_state=14)
```

如果你看下现有的数据集，你就会发现还需要做进一步的预处理。数据集有不少噪音，但最致命的问题（从数据分析角度看）是，它还包含其他用户所写的内容，回复邮件时会带上别人之前写的邮件。我们来看下邮件`documents[100]`：

I am disappointed on the timing but I understand. Thanks. Mark

-----Original Message-----

From: Greenberg, Mark

Sent: Friday, September 28, 2001 4:19 PM

To: Haedicke, Mark E.

Subject: Web Site

Mark -

FYI - I have attached below a screen shot of the proposed new look and feel for the site. We have a couple of tweaks to make, but I believe this is a much cleaner look than what we have now.

在这篇文档中，上面的邮件是对下面邮件的回复，这种情况很常见。第一部分是来自Mark Haedicke，而下面的邮件是Mark Greenberg写给Mark Haedicke。只有前面的内容（第一处“-----Original Message-----”之前）是发件人所写，这才是我们所关注的。

抽取这些信息不容易。邮件没有统一的格式。不同的邮件服务提供商使用不同的头部，对于回复内容有不同的定义形式，简直就是“为所欲为”4。就是在这样的环境中，电子邮件还被广泛使用，简直就是一个奇迹。

> 4读到此处，不禁想起各种浏览器的纷争。——译者注

当然还是有些共用的模式可以用。我们可以用`quotequail`包来查找邮件中的新内容，抛弃被回复的邮件及其他不相干信息。

你可以使用pip安装quotequail：pip3 install quotequail。

我们编写一个简单的函数封装quotequail的功能，方便调用它处理所有的文档。首先导入quotequail，声明函数。

```
import quotequail
def remove_replies(email_contents):
```

接着用quotequail把邮件解析为几个部分，返回字典结构。代码如下：

```
    r = quotequail.unwrap(email_contents)
```

有时候，r可能为none，比如邮件出于这样那样的原因解析失败。遇到这种情况，邮件原样返回。在处理真实数据集时，经常需要检测变量值是否为空。代码如下：

```
    if r is None:
        return email_contents
```

quotequail的返回结果中，我们真正需要的是text_top这部分。如果字典r中存在text_top，返回它的值。

```
    if 'text_top' in r:
        return r['text_top']
```

如果不存在，也就是quotequail没能找到该键。它也可能找到了text键，那返回text的值。代码如下：

```
    elif 'text' in r:
        return r['text']
```

最后，如果这两个键都没有找到，返回邮件全部内容，希望多少会对数据分析有点用处。

```
    return email_contents
```

对数据集中的每一封邮件，运行上述函数，做一遍预处理。

```
documents = [remove_replies(document) for document in documents]
```

我们之前看过的那封含有被回复邮件的文档，经过处理后，只包含Mark Greenberg所发的内容。

I am disappointed on the timing but I understand. Thanks. Mark

9.5.3 组装起来

我们可以使用前面实验所用到的参数空间和分类器——在新数据集上进行训练。默认情况下，scikit-learn会重新进行训练——后续调用fit()函数将会丢掉之前的信息。

> 还有一类算法叫作线上学习，它们使用新数据更新训练结果，但不是每次都重新进行训练。后续章节包括第10章新闻语料分类，我们会实际用到线上学习这类算法。

跟前面一样，我们使用cross_val_score计算正确率并输出结果。代码如下：

```
scores = cross_val_score(pipeline, documents, classes, scoring='f1')
print("Score: {:.3f}".format(np.mean(scores)))
```

F1值为0.523，对于包含这么多噪音的数据集来说，结果还算可以。添加更多数据（比如增加max_docs_author的值）应该会提升效果。

9.5.4 评估

一般来说，只凭借一个数字来评估效果不是个好主意，可能会忽略很多东西。拿F值来说，它考察的点就比较全面，那些分类结果好，却没有实际用处的雕虫小技也就不能蒙混过关。前面多次使用的正确率就有破绽，比如邮件过滤器把所有邮件归为垃圾邮件，也能达到80%的正确率，但是这种方法却没有实际用处。出于这个原因，我们还要对评估结果进行深入探讨。

我们先来看下混淆矩阵，第8章用神经网络破解验证码曾经用过。首

先，对测试集数据进行预测。前面代码使用cross_val_score，并没有给出可用的训练模型，因此我们需要重新训练一个模型。我们先来把语料切分为训练集和测试集。

```
from sklearn.cross_validation import train_test_split
training_documents, testing_documents, y_train, y_test =
train_test_split(documents, classes, random_state=14)
```

接着，把训练集传入流水线，进行训练，接着对测试集进行预测。

```
pipeline.fit(training_documents, y_train)
y_pred = pipeline.predict(testing_documents)
```

你可能想知道最好的参数组合是什么。我们可以方便地从网格搜索对象（流水线classifier这一步）中获取到这些参数组合。

```
print(pipeline.named_steps['classifier'].best_params_)
```

上述代码输出分类器的所有参数。然而大部分参数都使用默认值。只有C和kernel两个参数，我们使用网格搜索寻找合适的值，它们分别被设置为1和linear。

创建混淆矩阵：

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_pred, y_test)
cm = cm / cm.astype(np.float).sum(axis=1)
```

我们接下来作图要用到刻度，因此需要取到发件人。还好，加载数据集时，我们用字典来保存了发件人及其编号。代码如下：

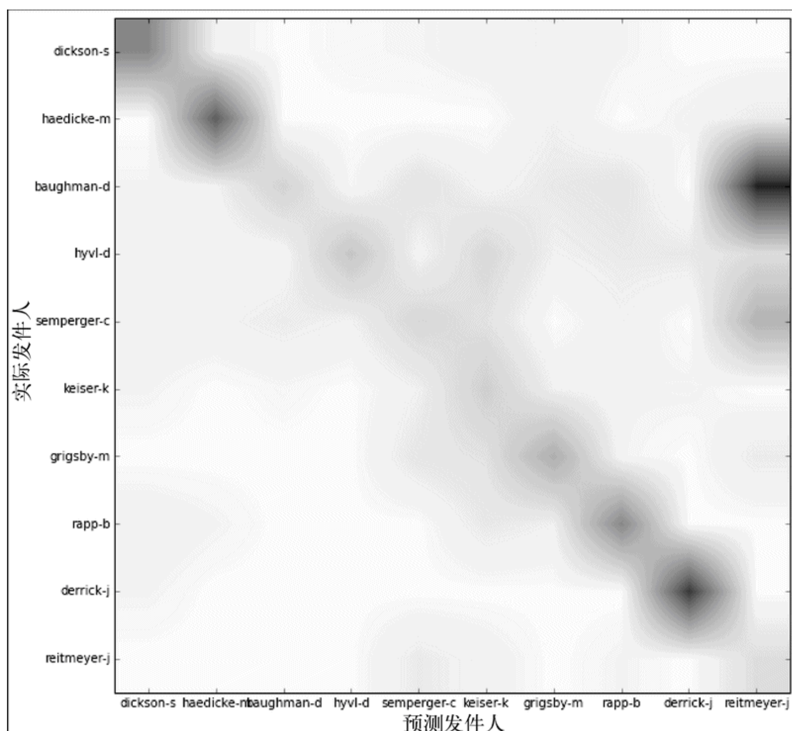
```
sorted_authors = sorted(authors.keys(), key=lambda x:authors[x])
```

最后，使用matplotlib绘制混淆矩阵。代码跟上章大同小异，不同的地方用粗体表示，只是把坐标轴刻度改为发件人。

```
%matplotlib inline
from matplotlib import pyplot as plt
plt.figure(figsize=(10,10))
```

```
plt.imshow(cm, cmap='Blues')
tick_marks = np.arange(len(sorted_authors))
plt.xticks(tick_marks, sorted_authors)
plt.yticks(tick_marks, sorted_authors)
plt.ylabel('Actual')
plt.xlabel('Predicted')
plt.show()
```

图像如下。



我们可以看到哪些发件人在大多数情况下都能正确预测——正确率高的形成了一条清晰的对角线。我们还能看出哪些发件人的邮件经常被弄混（颜色越深，弄混的次数越多）：例如，baughman-d的邮件经常被当作是reitmeyer-j发的。

9.6　小结

本章研究的是文本挖掘领域中的作者归属问题，我们分析了两类特征的效果：功能词和字符N元语法。我们在词袋模型中使用功能词——

提前选好的一组词，计算出这些词的词频。字符N元语法的处理流程跟基于单词的N元语法很相似，但是分析器转而关注字符而不是单词。此外，N元语法由一系列字符组成。单词N元语法由于可以提供单词的语境而没有增加多少计算量，在某些应用中也值得一试。

我们用SVM来进行分类，它通过找出使得两个类别之间间隔最大的线进行分类。线上的属于一类，线下的属于另外一类。跟其他分类任务一样，我们也需要有数据集（邮件）。

我们使用安然公司的邮件作为数据集，它包含很多噪音，比如包含被回复的邮件等。所以分类效果比起图书数据集要差不少。然而，在有10个发件人的情况下，半数以上邮件的发件人都能预测正确。

下一章，我们要研究的问题是没有目标类别的情况下，怎么对数据进行分类。这种叫作无监督学习，比起预测更像是探索性问题。我们要使用的依然是充满噪音的文本数据集。准备好接受挑战了吗？

第 10 章　新闻语料分类

前面大部分章节在对数据进行挖掘前，新数据所属的目标类别范围是已知的。在训练过程中，我们能够了解到变量跟类别之间存在怎样的关系。这种已知目标类别的学习任务，叫作有监督学习。本章研究的是在不知道类别的情况下如何进行数据挖掘。这叫作无监督学习，它偏重于探索、发现隐藏在数据里的信息，而不是用模型来分类，其探索性更强。

本章将介绍如何对新闻语料进行聚类，以发现其中的趋势和主题。本章还将讨论怎样从链接聚合网站抽取新闻语料。

本章主要介绍如下几个概念。

- 从任意新闻网站获取文本内容
- 使用reddit网站API采集有趣的新闻报道
- 使用聚类分析进行无监督的数据挖掘
- 抽取文档主题
- 用线上学习方法无需再次训练即可更新模型
- 组合不同的模型进行聚类

10.1　获取新闻文章

本章将构建一个按照主题为最新的新闻报道分组的系统。你可以运行几周（或更长时间）以了解这段时间新闻趋势的变化。

系统首先从流行的链接聚合网站reddit[1]寻找新闻报道的链接。reddit存储了大量其他网站的链接，还提供讨论区。网站收集的链接按照类别进行分类，这些类别被统称为subreddit，比如有电视节目、趣味图片等类别。本章关注的是新闻这一类链接。本章使用/r/worldnews类别的链接，但是我们所编写的代码也可以用来抓取其他类别的语料。

[1] Y Combinator孵化的项目。最初使用Lisp语言开发，后改用Python，网站开放源代码，

> 可以从Github上找到。创始人Steve Huffman在Udacity上开设了一门讲解Web开发的课程，课程编号为CS253。Steve卖掉reddit后，创立了专注于旅游搜索的Hipmunk网站。2015年再度回到reddit，出任CEO。web.py的作者Aaron Swartz也是reddit的联合创始人。——译者注

本章的目标是下载大家所关注的新闻报道，对其进行聚类分析，寻找其中的主题或主要概念，无需人工分析成百上千篇新闻报道，就能洞察人们关注的焦点。

10.1.1 使用Web API获取数据

前面有几章都是使用Web API从网站抽取数据。例如，第7章就用到Twitter网站的API。采集数据是数据挖掘流水线至关重要的一个环节。Web API是采集多个主题数据的绝佳工具。

使用Web API采集数据，有三个注意事项：授权方法、请求频率限制和API端点（endpoints）。

授权方法是数据提供方用来管理数据采集方的。数据提供方以此了解谁正在采集数据，确保采集方抓取数据的频率没有超出上限，同时对采集方都采集了哪些数据也做到心中有数。对于大多数网站，普通的个人用户账号就能用来采集数据，但也有部分网站要求采集方使用正式的开发者账号。

采集频率限制规定了采集方在约定时间内的最大请求次数，尤其是针对免费提供的服务。在使用数据获取接口时，一定要了解清楚，不同的网站很可能有着不同的规定。Twitter的API要求每15分钟（视调用的API而定）之内最多请求180次数据。reddit规定每分钟至多发起30次请求，下面我们还会讲到。其他有些网站会限制每天的请求次数，当然也有限制到秒级别的。即使同一个网站，不同的API调用，也有着截然不同的采集频率限制。例如，谷歌地图对每种资源API的调用限制，与按小时请求就有着不同规定，对前者限制更为严格。

> 如果你的应用或实验需要发起更多的请求和更快的响应，超出了免费API所能提供的，你可以寻求与API提供商进行商业合作。

API端点指的是用来抽取信息的实际网址。不同网站提供不同的接口，大部分Web API提供RESTful接口（Representational State Transfer，表述性状态转移）。RESTful接口通常与HTTP协议使用相同的操作：GET、POST、DELETE是最常用的。例如，使用下述API端点检索某一资源的相关信息：www.dataprovider.com/api/

resource_type/resource_id/。

获取信息，只需要发送HTTP GET请求到指定的网址。服务器返回资源信息、信息类型和ID。大多数API都是按照这种结构进行封装的，虽然在具体实现上会有所差异。大多数提供API的网站，都会给出详细的文档，列出所有开放使用的API的具体参数。

我们来看下获取信息的具体步骤。首先，设置连接reddit网站所用到的参数，这里需要用到reddit开发者密钥。从https://www.reddit.com/login登录reddit网站，打开https://www.reddit.com/prefs/apps。点击“are you a developer? create an app...”，填写表单，选择`script`类型。你就能得到用户ID和密钥，新建一个笔记本文件，输入以下代码。

```
CLIENT_ID = "<Enter your Client ID here>"
CLIENT_SECRET = "<Enter your Client Secret here>"
```

reddit还要求你在使用他们的API时，设置唯一的用户代理（user agent）字符串。在设置用户代理时，注意把自己的reddit用户名也写上，以创建唯一标识你应用的用户代理字符串。我的用户代理字符串中包含本书的英文名、章节编号“chapter 10”及版本号“0.1”，你可以使用其他内容。如果你的用户代理跟别人重复，你的请求频率将受到很大限制。

```
USER_AGENT = "python:<your unique user agent> (by /u/<your reddit username
```

此外，你需要使用自己的用户名和密码登录reddit。如果你还没有账号，请注册一个（免费且无需验证个人信息）。

> 下一步会用到你的密码，把代码分享给别人之前，记得删除它。如果你不打算在代码中写入密码，把密码设置为none，这样每次运行时输入密码即可。但是，你需要在启动笔记本文件的终端输入，而不是在笔记本文件里输入，这是由笔记本文件的工作方式决定的。如果你无法在终端输入，请直接在代码里设置好密码。IPython的开发人员正在编写修复这个问题的插件，但是写作本书时，插件还没写好。

指定用户名和密码，代码如下：

```
USERNAME = "<your reddit username>"
PASSWORD = "<your reddit password>"
```

接着，创建登录函数，使用以上信息进行登录。reddit的登录接口将会返回进一步连接所需要的令牌，这也就是登录函数的返回结果。函数声明如下：

```
def login(username, password):
```

首先，如果你不想把密码写到代码里，把它设置为none，但是你需要在命令行输入，前面已经说过。代码如下：

```
    if password is None:
        password = getpass.getpass("Enter reddit password for user {}:
".format(username))
```

使用独一无二的用户代理很重要，否则你的连接将严重受限。代码如下：

```
    headers = {"User-Agent": USER_AGENT}
```

接着，创建HTTP授权对象，以登录reddit服务器。

```
    client_auth = requests.auth.HTTPBasicAuth(CLIENT_ID, CLIENT_SECRET)
```

发起POST请求到access_token端点以实现登录，POST的数据有用户名、密码、授权类型。本例中授权类型为使用密码进行登录。

```
    post_data = {"grant_type": "password", "username": username, "passworc
```

最后，函数使用requests库发起登录请求（通过HTTP POST请求实现），POST请求返回的结果为字典类型。其中有一项就是进一步请求所需的令牌。代码如下：

```
    response = requests.post("https://www.reddit.com/api/v1/access_token",
    return response.json()
```

调用上述登录函数，获取令牌。

```
token = login(USERNAME, PASSWORD)
```

令牌对象为一字典结构，其中的access_token键对应的值就是进一步请求所需的令牌。该对象还包含令牌的使用范围（全局）和过期时间。例如：

```
{'access_token': '<semi-random string>',  'expires_in': 3600,
'scope': '*',  'token_type': 'bearer'}
```

10.1.2　数据资源宝库reddit

链接聚合网站reddit（www.reddit.com）拥有几亿用户，虽然英文版主要面向美国。每个用户都可以发布他们感兴趣的网站的链接，同时为该链接指定标题。其他用户对其点赞，表示他们喜欢这个链接的内容，也可以投反对票，表示不喜欢。点赞最高的链接将被移到网页的最上面，没有多少人喜欢的将不会显示。随着时间的推移，先前发表的链接也将不再显示（根据喜欢数来定）。分享的链接被点赞后，用户将会获得叫作karma2的积分，这也是为了激励用户只分享好故事。

> 2karma，指因果报应。reddit网站使用该词，应是取其善有善报之意。——译者注

reddit用户也可以发表链接之外的其他内容，这些内容叫作自己的广播3，它由用户输入的标题和正文组成，通常是为了提问或是引发讨论，不在积分赚取范围之内。本章仅关注链接类广播，不关注评论类的。

> 3广播对应英文单词post，指的是用户发表的一条内容。——译者注

网站的所有广播被分到叫作subreddit的不同栏目，每个栏目包含一系列与之相关的广播。用户发布广播时，需要选择广播所属的栏目。每个栏目都有自己的管理员和内容管理规则，用户不能发表与栏目无关的内容。

广播默认根据热度（Hot）进行排序，一条广播的热度由其存在的时间、喜欢数、不喜欢的次数来决定。除了热度外，还有最新发表的（New）和一定时间内最受欢迎的（Top）两种排序方法，最新广播

可能包含大量的垃圾信息。本章将按照热度来选取广播，这些广播比较新，内容质量也比较高（单纯根据时间排序，得到的链接有很多是垃圾链接）。

使用我们前面获取到的令牌，就可以获取一个栏目的一系列链接。/r/<subredditname> API端点默认返回所指定栏目的热门文章。我们把栏目指定为世界新闻/r/worldnews。

```
subreddit = "worldnews"
```

我们使用字符串格式化方法，用刚指定的栏目来创建完整的URL。

```
url = "https://oauth.reddit.com/r/{}".format(subreddit)
```

接下来，需要设置头部，这样才能指定授权令牌，设置独一无二的用户代理，争取不会被过分限制请求次数。代码如下：

```
headers = {"Authorization": "bearer {}".format(token['access_token']),
           "User-Agent": USER_AGENT}
```

然后，跟之前一样，使用requests库发起请求，别忘了指定头部。

```
response = requests.get(url, headers=headers)
```

调用response的json方法，将会得到包含reddit返回信息的一个Python字典。它包含给定栏目的25条广播，对它们进行遍历，输出每条广播的标题。广播的相关内容存储在字典键为data的那一项中。代码如下：

```
for story in result['data']['children']:
    print(story['data']['title'])
```

10.1.3 获取数据

我们数据集的每条广播都来自/r/worldnews栏目的热度列表。上节讲解了连接reddit网站和抽取广播内容的方法。我们来创建一个函数，把这两个步骤整合起来，抽取指定栏目每条广播的标题、链接和

喜欢数。

我们遍历世界新闻栏目，最多一次获取100篇新闻报道。我们还可以使用分页，reddit最多允许我们读多少页，我们就读多少页。但是我们这里最多读5页。

因为我们代码会重复调用同一个API，为了防止超出网站所规定的采集频率限制，我们可以使用sleep函数，先来导入它。

```
from time import sleep
```

接下来要定义的这个函数接收三个参数，分别为栏目名称、授权令牌和读取的页数，默认值为5。

```
def get_links(subreddit, token, n_pages=5):
```

创建列表，存储新闻报道。

```
    stories = []
```

我们在第7章见过如何在Twitter的API中使用分页功能。Twitter在返回结果时，同时返回一个游标，调用接口时，再一并传递游标，Twitter再利用此游标获取当前结果的下一页。reddit的API游标用法几乎和Twitter一致，只不过它使用after参数。获取第一页数据时用不到这个参数，将其置为none，获取到第一页后，再给它传一个有意义的值。代码如下：

```
    after = None
```

遍历我们想得到的那些页。

```
    for page_number in range(n_pages):
```

在上述循环中，设置好头部和URL，方法跟之前一样。

```
        headers = {"Authorization": "bearer
        {}".format(token['access_token']),
```

```
        "User-Agent": USER_AGENT}
    url = "https://oauth.reddit.com/r/{}?limit=100".
        format(subreddit)
```

从第二次循环开始，我们需要设置after参数（否则返回结果都一样）。下一次循环所用到的after的值在前一次循环中设置。如果after值不为none，把它添加到URL的末尾，告诉reddit我们接下来要获取哪页的数据。代码如下：

```
    if after:
        url += "&after={}".format(after)
```

下面跟之前一样，使用requests库发起请求，在返回结果上调用json()方法，得到Python字典。

```
    response = requests.get(url, headers=headers)
    result = response.json()
```

返回结果中包含下一轮迭代所需的after值，用它来更新after参数。

```
    after = result['data']['after']
```

暂停两秒钟，防止超出API使用频率限制。

```
    sleep(2)
```

在循环体的最后，从reddit的返回结果中获取到每篇报道，把它们添加到stories列表中。我们仅需要标题、URL和喜欢数这三项数据。代码如下：

```
    stories.extend([(story['data']['title'], story['data']['url'],
story['data']['score'])
                    for story in result['data']['children']])
```

最后（循环体外面），返回找到的所有新闻报道。

```
return stories
```

```

```

调用get_links函数时传入栏目名称和授权令牌即可。

```
stories = get_links("worldnews", token)
```

返回结果包含500篇新闻报道的标题、URL等，我们接下来就可以获取到这些URL所指向页面的文本内容。

10.2 从任意网站抽取文本

我们从reddit收集到的网址所指向的网站分属不同的网站组织。这些网站的目标用户是普罗大众而不是计算机程序。当我们尝试用程序获取里面的实际内容时，可能会遇到种种困难，因为如今的网站，有很多逻辑是在后台运行：调用JavaScript库，应用样式表，用AJAX加载广告，在侧边栏增加很多内容等，这些功能增加了网站的复杂程度。这些技术的应用使得当今的Web看起来鲜活、生动、丰富多彩，却增加了自动采集信息的难度。

10.2.1 寻找任意网站网页中的主要内容

我们首先需要访问每个链接，下载各个网页，将它们保存到Data文件夹中事先建好的用于存放原始网页的文件夹raw。后面，我们就要从这些原始网页中获取有用的信息。先把全部网页都保存下来，比起后面时不时地下载网页方便多了。首先，指定存放原始网页的目录。

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data",
"websites", "raw")
```

后面我们要用MD5散列算法为每篇报道创建一个唯一的文件名，所以先导入hashlib。散列函数将输入（包含新闻报道名称的字符串）转换为一个看上去像是随机产生的字符串。对于相同的输入，散列函数返回相同的结果，但是不同输入之间的微小差别将会导致截然不同的输出结果。并且，散列函数为单向函数，根据散列值无法得到原来的值。导入hashlib，代码如下：

```
import hashlib
```

对于网页下载失败的网站，我们直接跳过。为了避免错过太多网页，我们维护一个计数器，统计下载失败的次数。如果是系统本身的问题阻止下载，那我们就要解决这些问题。如果失败次数过多，我们就要找出到底是什么东西在作怪，尝试去解决问题。例如，如果计算机没有联网，所有的500次下载都会失败，你得先解决联网问题，才能再开展下面的工作！

如果所有网页都能成功下载，计数器输出值应该为0。

```
number_errors = 0
```

接下来，遍历每一篇新闻报道。

```
for title, url, score in stories:
```

对报道的标题进行散列操作，作为输出文件名，以保证唯一性。因为reddit网站不要求文章标题具有唯一性，因此两篇报道可能使用相同的标题，这就会导致数据集中两条数据之间的冲突。我们使用MD5算法对报道的URL进行散列操作。现在人们发现MD5也不是绝对可靠的，但是就我们这么小的数据规模来说不太可能出问题（出现碰撞情况），即使出现这样的问题，也不用太担心。

```
    output_filename = hashlib.md5(url.encode()).hexdigest()
    fullpath = os.path.join(data_folder, output_filename + ".txt")
```

接下来下载网页，保存到输出文件夹中。

```
    try:
        response = requests.get(url)
        data = response.text
        with open(fullpath, 'w') as outf:
            outf.write(data)
```

如果下载网页时出现问题，跳过问题网站，继续下载下一个网站的网页。上述代码能够完成我们收集到的95%的网站下载任务，对本章的应用来说足够了，因为我们寻找的是大体趋势而不是做精准研究。有时我们必须下载到所有网站的内容，这时就得调整代码，以处理各种可能的错误。抓取失败的5%～10%的网站，需要编写更为复杂的代码才能进行处理。我们来捕获出现的错误（这可是因特网，会有很多

意想不到的错误），增加计数器计数，继续下载下个网站的网页。

```
    except Exception as e:
        number_errors += 1
        print(e)
```

如果错误很多，把上面print(e)那一行改为raise，调用异常中断机制，以便修改问题。

现在，我们原始网页文件夹raw中有很多网页，查看这些网页（用文本编辑器打开），你会发现新闻报道的内容湮没在HTML、JavaScript、CSS以及其他内容之中。因为我们只对报道本身感兴趣，就需要一种从不同网站的网页中抽取内容的方法。

10.2.2　组装起来

获得原始网页后，我们需要找出每个网页中的新闻报道内容。有一些在线资源使用数据挖掘方法来解决这个问题，资源列表请见附录。一般来说，很少需要用到这些复杂的算法，当然使用它们能得到更为精确的结果。这也是数据挖掘艺术的一部分——知道什么时候用，什么时候不用。

首先，获取到raw文件夹中的所有文件名。

```
filenames = [os.path.join(data_folder, filename)
             for filename in os.listdir(data_folder)]
```

接着，创建输出文件夹，新闻报道内容抽取出来后，将保存到该文件夹下。

```
text_output_folder = os.path.join(os.path.expanduser("~"), "Data",
                                  "websites", "textonly")
```

接着，编写从网页中抽取文本的代码。我们将使用lxml包解析HTML文件，lxml的HTML解析器容错能力强，可以处理不规范的HTML代码。导入语句如下所示：

```
from lxml import etree
```

文本抽取分为以下三步：首先，遍历HTML文件的每一个节点，抽取其中的文本内容。其次，跳过JavaScript、样式和注释节点，这些节点不太可能包含对我们有价值的信息。最后，确保文本内容长度至少为100个字符。分析文章主题，100个字符就够用，但是要想得到更准确的结果，文章长度有待增加。

前面说过，我们对脚本、样式或注释不感兴趣。因此，创建列表，存放这些不可能包含新闻报道内容的节点。代码如下：

```
skip_node_types = ["script", "head", "style", etree.Comment]
```

我们创建一个函数，把HTML文件解析成`lxml etree`对象，然后创建另外一个函数，解析前面得到的树，寻找文本。第一个函数简单易懂：打开网页文件，用`lxml`库的`parse`方法解析HTML文件。代码如下：

```
def get_text_from_file(filename):
    with open(filename) as inf:
        html_tree = lxml.html.parse(inf)
    return get_text_from_node(html_tree.getroot())
```

上述函数的最后一行，调用`getroot()`函数，获取到树的根节点，而不是整棵树`etree`，这样文本抽取函数`get_text_from_node`以节点作为参数，它能处理包括根节点在内的所有节点，便于递归调用。

文本抽取函数将在任何子节点上调用自己，以抽取子节点中的文本内容，最后返回拼接在一起的所有子节点的文本。

如果一个节点没有任何子节点，文本抽取函数返回该节点的文本内容。如果该节点不包含任何内容，就返回空字符串。请注意，还需要执行上面提到的文本抽取的第三步——检查文本长度是否达到100个字符。代码如下：

```
def get_text_from_node(node):
    if len(node) == 0:
        # No children, just return text from this item
        if node.text and len(node.text) > 100:
            return node.text
        else:
            return ""
```

明确节点有子节点之后，就对每一个子节点递归调用文本抽取函数，把返回的文本内容拼接在一起。代码如下：

```
    results = (get_text_from_node(child) for child in node
                     if child.tag not in skip_node_types)
    return "\n".join(r for r in results if len(r) > 1)
```

上面return语句中的if语句是为了防止返回空行（例如，有的节点没有子节点和文本内容）。

遍历所有的原始网页，从文件里抽取文本，把结果保存到纯文本文件4夹中。

> 4指去除掉网页代码后剩下的文本内容。——译者注

```
for filename in os.listdir(data_folder):
    text = get_text_from_file(os.path.join(data_folder, filename))
    with open(os.path.join(text_output_folder, filename), 'w')
      as outf:
        outf.write(text)
```

打开纯文本文件夹中的文件，查看它们是否有内容。如果很多都不包含新闻报道内容，提升最少字符限制，我们刚才用的是100。如果设置为更高的值，仍不起作用，或者你的应用对抽取文本内容有更高的要求，请尝试使用附录介绍的更为复杂的方法。

10.3　新闻语料聚类

本章的目的是通过聚类发现新闻语料中潜藏的趋势。我们将使用诞生于1957年的经典机器学习算法k-means（k均值）。

聚类属于无监督学习，我们使用聚类算法探索隐藏在数据里的奥秘。我们的数据集由大约500篇新闻报道组成，人工查看这些文章的主题费时费力。即使使用概括统计方法也不容易，对这种方法而言数据还是相当多。而聚类分析则可以按照主题把它们分成不同的簇，然后就可以按簇研究它们的主题。

我们在没有明确的类别的情况下会使用聚类方法。从这个意义上来讲，聚类算法学习时没有明确的方向性，它们根据目标函数而不是数据潜在的含义来学习。因此，选择聚类效果好的特征就很有必要。对

于有监督学习，即使你选用了效果较差的特征，学习算法可以选择不用这些特征。例如，支持向量机为对分类用处不大的特征分配很小的权重。然而，聚类算法会综合所有特征给出最后结果——即使那些不会提供给我们答案的特征。

在对真实的数据进行聚类分析前，最好了解哪些特征适用于当前任务。本章使用词袋模型。我们寻找的是主题相关的簇，因此使用主题相关的特征为数据建模。我们之所以知道这些特征可用来聚类，是因为别人用有监督方法解决类似问题时用过这些特征。对比来说，如果要按照作者把作品分组，我们就要使用类似第9章用到的特征。

10.3.1　k-means算法

k-means聚类算法迭代寻找最能够代表数据的聚类质心点。算法开始时使用从训练数据中随机选取的几个数据点作为质心点。k-means中的*k*表示寻找多少个质心点，同时也是算法将会找到的簇的数量。例如，把*k*设置为3，数据集所有数据将会被分成3个簇。

k-means算法分为两个步骤：为每一个数据点5分配簇标签，更新各簇的质心点。

> 5一个数据点对应数据集中一条数据，把数据集看成样本，一条数据即可被称作是一个个体。——译者注

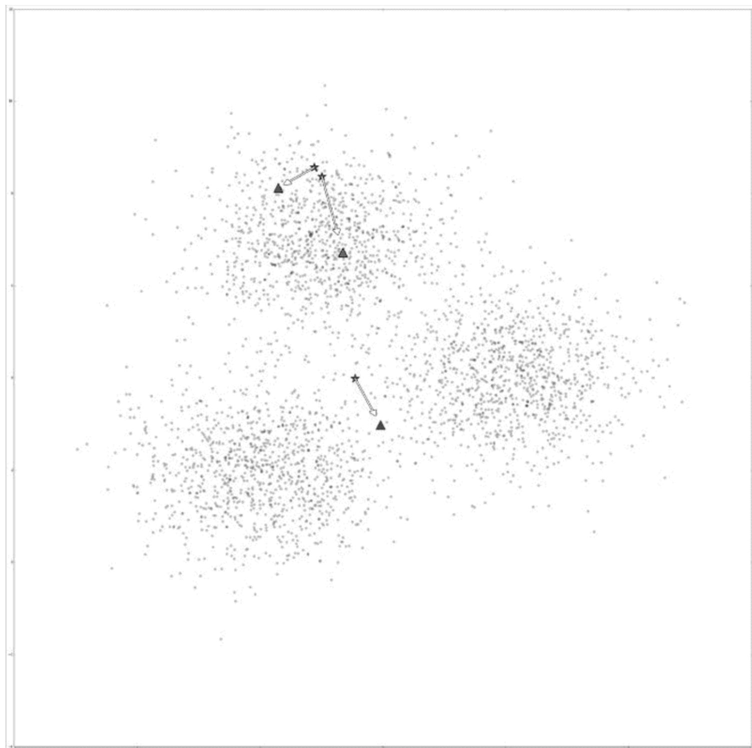
分簇这一步中，我们为数据集的每个个体设置一个标签，把它和最近的质心点联系起来。对于距离质心点1最近的个体，我们为它们分配标签1，距离质心点2最近的个体，分配标签2，以此类推。标签相同的个体属于同一个簇，所有带有标签1的数据点属于簇1（只是暂时的，数据点所属的簇会随着算法的运行发生变化）。

更新环节，计算各簇内所有数据点的均值，更新质心点。

k-means算法会重复上述两个步骤；每次更新质心点时，所有质心点将会小范围移动。这会轻微改变每个数据点在簇内的位置，从而引发下一次迭代时质心点的变动。这个过程会重复执行直到条件不再满足时为止。通常是在迭代一定次数后，或者当质心点的整体移动量很小时，就可以终止算法的运行。有时可以等算法自行终止运行，这表明簇已经相当稳定——数据点所属的簇不再变动，质心点也不再改变时。

下图表示的是对随机创建的数据集执行k-means算法，数据集包含三

个簇。图中的三颗五角星表示最初随机从数据集中选取的三个质心点。k-means算法经过5轮迭代后，质心点发生了改变，具体位置用三角形来表示。



k-means算法因它所使用的数学方法和久经考验而充满魅力。该算法只有（大体上可以这么说）一个参数，虽然距它提出至今已过了半个多世纪，但由于它在很多数据挖掘问题上效果很好，仍被频繁使用。

scikit-learn实现了k-means算法，直接从cluster模块导入即可。

```
from sklearn.cluster import KMeans
```

我们顺便把CountVectorizer类的好兄弟TfidfVectorizer导进来。这个向量化工具根据词语出现在多少篇文档中，对词语计数进行加权。出现在较多文档中的词语权重较低（用文档集数量除以词语出现在的文档的数量，然后取对数）。对于很多文本挖掘应用，使用该种权重计算方法，能够有效提升效果。代码如下：

```
from sklearn.feature_extraction.text import TfidfVectorizer
```

创建数据分析流水线，流水线分两步，第一步是特征抽取，第二步是调用k-means算法。代码如下：

```
from sklearn.pipeline import Pipeline
n_clusters = 10
pipeline = Pipeline([('feature_extraction', TfidfVectorizer(max_
df=0.4)),
                     ('clusterer', KMeans(n_clusters=n_clusters))
                     ])
```

我们为参数max_df设置了一个很低的值0.4，表示忽略出现在**40%**及以上的文档中的词语。这个参数可用来剔除本身不含有主题相关含义的词语。

> 删除在40%及以上文档中出现的词语将会删除功能词，这种预处理对于第9章将有害无益。

首先训练算法，然后再用它来做预测。前几章的分类任务都是按照这两个步骤实施，但是这里有一点不同——我们没有为fit函数指定目标类别。这也正是无监督学习任务的含义所在！代码如下：

```
pipeline.fit(documents)
labels = pipeline.predict(documents)
```

变量labels包含每个数据点的簇标签。标签相同的数据点属于同一个簇。需要指出的是簇标签本身没有含义：不能说簇1和簇2比簇1和簇3更相似。

我们可以使用Counter类来查看每个簇有多少数据点。

```
from collections import Counter
c = Counter(labels)
for cluster_number in range(n_clusters):
    print("Cluster {} contains {} samples".format(cluster_number,
c[cluster_number]))
```

聚类问题的结果（注意你的数据集可能跟我的不同）往往是一个簇很大，包含了大多数个体，再有几个中等规模的簇，最后还有几个只包

含一两个个体的很小的簇，这种不平衡性很常见。

10.3.2 评估结果

聚类分析主要是探索性分析，因此很难有效地评估聚类算法结果的好坏。评估算法结果最直接的方式是根据它要学习的标准对其进行评价。

> 如果你有测试集，你可以对其进行聚类分析来评价效果。更多细节请见http://nlp.stanford.edu/IRbook/html/htmledition/evaluation-of-clustering-1.html。

对于k-means算法，寻找新质心点的标准是，最小化每个数据点到最近质心点的距离。这叫作算法的惯性权重（inertia），任何经过训练的KMeans实例都有该属性。

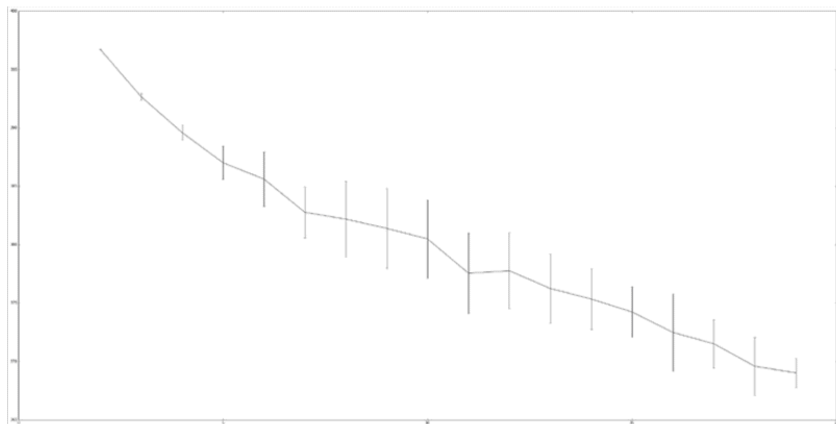
```
pipeline.named_steps['clusterer'].inertia_
```

输出结果为343.94，不过这个值本身没有意义，但是可以用它来确定分为多少簇合适。前面的例子，n_clusters的值被设置为10，但这是最佳值吗？下面代码n_clusters依次取2到20之间的值，每取一个值，k-means算法运行10次。每次运行算法都记录惯性权重。

对于n_clusters变量的每个取值，仅训练X矩阵一次，以（极大）提升代码运行速度。

```
inertia_scores = []
n_cluster_values = list(range(2, 20))
for n_clusters in n_cluster_values:
    cur_inertia_scores = []
    X = TfidfVectorizer(max_df=0.4).fit_transform(documents)
    for i in range(10):
        km = KMeans(n_clusters=n_clusters).fit(X)
        cur_inertia_scores.append(km.inertia_)
    inertia_scores.append(cur_inertia_scores)
```

变量inertia_scores存储了n_clusters取2到20每个值时所对应的惯性权重。我们把惯性权重和簇的数量做成图，以便了解它们之间的关系。



整体而言，随着簇数量的增加，质心点和其他数据点位置的调整逐渐减少，惯性权重应该逐渐降低，这是很容易就能从上图得到的结论。从6簇进一步分为7簇时，质心点是随机选取的，这将会直接影响到最终结果。尽管如此，整体的趋势（你的结果可能会有所不同）是簇数量为6时，惯性权重进行了最后一次大的调整。

随后惯性权重改变很小，虽然没有明确的标准可言。这样的时刻被称作是拐点（elbow），用图来表示就是曲线的顶点，看起来就像是肘部。有些数据集拐点很明显，但是有的数据集可能没有拐点（它们的图像看起来很平滑）。

根据上述分析，把`n_clusters`的值设置为6，重新运行算法。

```
n_clusters = 6
pipeline = Pipeline([('feature_extraction',
  TfidfVectorizer(max_df=0.4)),
                     ('clusterer', KMeans(n_clusters=n_clusters))
                     ])
pipeline.fit(documents)
labels = pipeline.predict(documents)
```

10.3.3　从簇中抽取主题信息

现在我们把关注点放到各簇上，尝试从中找到每个簇的主题。首先，从特征提取这一步抽取词表。

```
terms = pipeline.named_steps['feature_extraction'].get_feature_names()
```

计算每簇所包含的个体数量。

```
c = Counter(labels)
```

遍历所有的簇，输出每簇所包含的个体数量。评估结果时，要把簇的大小考虑进去——有的簇可能只有一个个体，因此不能代表新闻趋势。代码如下：

```
for cluster_number in range(n_clusters):
    print("Cluster {} contains {} samples".format(cluster_number,
c[cluster_number]))
```

接下来（在循环体内部），遍历该簇最重要的词语。首先，从质心点找出特征值最大的5个特征。代码如下：

```
    print("  Most important terms")
    centroid = pipeline.named_steps['clusterer'].cluster_centers_
[cluster_number]
    most_important = centroid.argsort()
```

依次输出这5个特征。

```
    for i in range(5):
```

对`i`取反，因为`most_important`数组中的最小值排在最前面。

```
        term_index = most_important[-(i+1)]
```

然后输出序号、词语和得分。

```
        print("  {0}) {1} (score: {2:.4f})".format(i+1, terms[term_
index], centroid[term_index]))
```

结果能较好地反映当时人们关注的焦点。我得到的结果（2015年3月）中，几个簇的主题为健康问题、中东紧张局势、朝鲜半岛紧张局势和俄罗斯相关问题。这些充斥着当时的新闻头条——虽然很多年以来就一直这样！

10.3.4　用聚类算法做转换器

另外提一下，k-means算法（以及其他聚类算法）还有比较有趣的一个特点，就是可以用来简化特征。特征简化的方法有很多种，比如主要成分分析（Principle Component Analysis）、潜在语义索引（Latent Semantic Indexing）等，这些方法还能用来创建新特征，但是它们通常对计算能力要求很高。

在前面的例子中，词表共有14 000个词条——这个数据集很大。我们的聚类算法把它们仅归为6个簇。我们可以使用数据点到质心点的距离作为特征创建一个特征数较之前少得多的数据集。

在k-means实例上调用转换函数。因为流水线最后一步是`k-means`实例，因此可以在它上面调用转换函数。

```
X = pipeline.transform(documents)
```

上面代码在流水线最后一步的`k-means`实例上调用转换方法。得到的矩阵有六个特征，数据量跟文档的长度相同。

你可以对得到的矩阵进行二次聚类，如果有目标类别的话，也可以用来分类。大致的流程是使用标注好的数据选取特征，用聚类方法把特征数减少到更容易操作的范围内，再用SVM等算法对前面处理过的数据集进行分类。

10.4　聚类融合

第3章研究了如何把几棵效果相对较差的决策树分类器整合在一起形成随机森林，用来处理分类任务。聚类算法也可以进行融合，这样做的主要原因是，融合后得到的算法能够平滑算法多次运行所得到的不同结果。正如我们之前所说，多次运行k-means算法得到的结果因最初选择的质心点不同而不同。多次运行算法，综合考虑所得到的多个结果，可以减少波动。

聚类融合方法还可以降低参数选择对最终结果的影响。大多数聚类算法对参数选择很敏感,参数稍有不同将带来不同的聚类结果。

10.4.1　证据累积

最基本的融合方法是对数据进行多次聚类，每次都记录各个数据点的簇标签。然后计算每两个数据点被分到同一个簇的次数。这就是证据累积算法（Evidence Accumulation Clustering，EAC）的精髓。

证据累积算法两大步骤如下：第一步，使用k-means等低水平的聚类算法对数据集进行多次聚类，记录每一次迭代两个数据点出现在同一簇的频率，将结果保存到共协矩阵（coassociation）中。第二步，使用另外一种聚类算法——分级聚类对第一步得到的共协矩阵进行聚类分析。分级聚类一个比较有趣的特性是，它等价于寻找一棵把所有节点连接到一起的树，并把权重低的边去掉。

遍历所有标签，记录具有相同标签的两个数据点的位置，创建共协矩阵。我们需要用到SciPy的稀疏矩阵csr_matrix。

```
from scipy.sparse import csr_matrix
```

注意函数声明中的参数为所有数据点的标签。

```
def create_coassociation_matrix(labels):
```

记录具有相同标签的两个数据点的行号和列号，把它们分别存储到rows和cols列表中。稀疏矩阵通常由一系列记录非零值位置的列表组成，csr_matrix就是这种类型的。

```
    rows = []
    cols = []
```

标签去重后，再进行遍历。

```
    unique_labels = set(labels)
    for label in unique_labels:
```

查找具有某一标签的所有数据点。

```
        indices = np.where(labels == label)[0]
```

对于每组标签相同的数据点，记录它们的位置。代码如下：

```
        for index1 in indices:
            for index2 in indices:
                rows.append(index1)
                cols.append(index2)
```

在所有循环的外面创建数据集，两个数据点标签相同时，数据集中该位置的值为1。有多少组数据点标签相同，数据集中就有多少个元素为1。代码如下：

```
    data = np.ones((len(rows),))
    return csr_matrix((data, (rows, cols)), dtype='float')
```

调用上述函数，传入所有的数据点标签，就能得到共协矩阵。

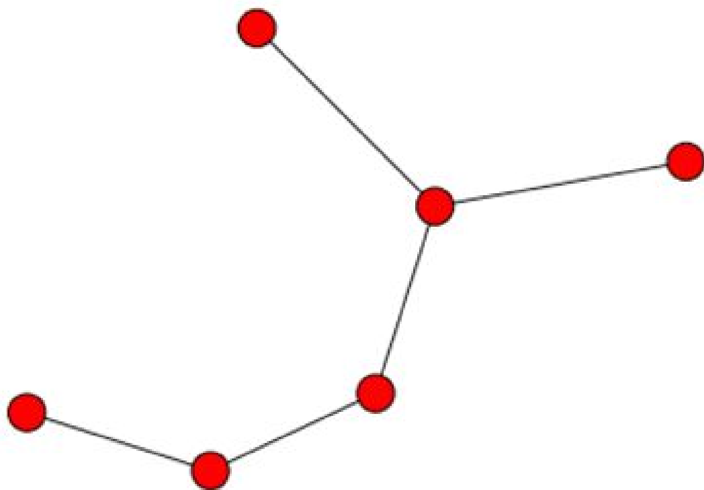
```
C = create_coassociation_matrix(labels)
```

现在我们可以把这些矩阵的多个实例整合起来，多次运行k-means算法，把结果结合起来看。运行C（在记事本新格子里输入C，并运行），查看有多少个非零值。从得到的结果来看，矩阵中大约半数的项有值，我的聚类结果有一个大的簇（各簇的大小越均匀，非零值数量越少）。

下一步对共协矩阵进行分级聚类。我们需要找到该矩阵的最小生成树，删除权重低于阈值的边。

从图的理论角度看，生成树为所有节点都连接到一起的图。最小生成树（Minimum Spanning Tree，MST）即总权重最低的生成树。结合我们的应用来讲，图中的节点对应数据集中的个体，边的权重对应两个顶点被分到同一簇的次数——也就是共协矩阵所记录的值。

下图的MST有6个节点。图中的节点可以在MST中多次使用，唯一的要求是所有的节点应该连接在一起。



我们使用SciPy sparse包的minimum_spanning_tree函数计算MST。

```
from scipy.sparse.csgraph import minimum_spanning_tree
```

可以直接在coassociation函数返回的稀疏矩阵上调用mst函数。

```
mst = minimum_spanning_tree(C)
```

然而，矩阵C中，值越高表示一组数据点被分到同一簇的次数越多——这个值表示相似度。相反，minimum_spanning_tree函数的输入为距离，高的值反而表示相似度越小。因此，我们对C取反再计算最小生成树。

```
mst = minimum_spanning_tree(-C)
```

上述函数返回结果为一个矩阵，大小跟C相同（行列数分别跟数据集样本数量和特征数相同），只不过保留了最小生成树中的边，其他都被删除了。

然后我们删除其边的权重小于阈值的节点。方法是，遍历MST矩阵中的每一条边，删除低于规定值的边。仅对共协矩阵（值为1或0，没有什么可处理的）进行一次遍历无法完成上述任务。因此，我们来创建额外的标签，创建一个新共协矩阵，然后把这两个矩阵相加。代码如下：

```
pipeline.fit(documents)
labels2 = pipeline.predict(documents)
C2 = create_coassociation_matrix(labels2)
C_sum = (C + C2) / 2
```

然后再计算MST，删除在这两个矩阵中都没有出现的边。

```
mst = minimum_spanning_tree(-C_sum)
mst.data[mst.data > -1] = 0
```

我们需要移除在C1和C2都没有出现的边——也就是值为1的。然而我们刚才在计算时，对`C_sum`取反，因此，阈值也应该使用相反数。

最后，我们找到所有的连通分支，也就是寻找移除低权重的边以后仍然连接在一起的节点。返回的第一个值为连通分支的数量（也就是有多少个簇），第二个值为每个数据点的标签。代码如下：

```
from scipy.sparse.csgraph import connected_components
number_of_clusters, labels = connected_components(mst)
```

我的数据集最终找到了8个簇，每个簇都跟之前差不多。这不足为奇，因为我们仅进行了两轮 k-means迭代，更多的迭代（请见下节）结果差异会更明显。

10.4.2　工作原理

k-means算法不考虑特征的权重，实际上它假定所有的特征取值范围相同。我们在第2章探讨过使用取值范围不同的特征所带来的问题。k-means算法寻找的是圆形簇（circular clusters），如下图所示。



从上图中，我们可以看到不是所有的簇都是圆形。左侧的簇呈圆形，这也是k-means算法很擅长处理的。中间为椭圆形，通过特征规范化（feature scaling），k-means算法可以处理该种聚类问题。最后一幅图甚至不是凸形的——这种形状很奇怪，用k-means算法来处理聚类结果为这种形状的数据集有难度。

证据累积算法的工作原理为重新把特征映射到新空间，上节讲到用k-means来做特征简化，本质上，证据累积算法使用与其相似的原则，每次运行k-means算法都相当于使用转换器对特征进行一次转换。但是我们这里仅使用实际的标签而不是到每个质心点的距离。我们使用存储在共协矩阵中的数据。

证据累积算法只关心数据点之间的距离而不是它们在原来特征空间的位置。对于没有规范化过的特征，仍然存在问题。因此，特征规范很重要，无论如何都要做（我们用`tf-idf`规范特征值，从而使特征具有相同的值域）。

我们在第9章见过使用SVM内核做类似的转换。这些转换方法很强大，在处理复杂数据集时记得使用。

10.4.3　实现

现在我们来把前面这些整合起来，创建接口跟`scikit-learn`对得上的简易聚类算法，实现证据累积的两个步骤。首先，使用`scikit-learn`的`ClusterMixin`创建证据累积算法类。

```
from sklearn.base import BaseEstimator, ClusterMixin
class EAC(BaseEstimator, ClusterMixin):
```

参数分别为第一步k-means算法运行次数（创建共协矩阵），用来删除边的阈值和每次运行k-means算法要找到的簇的数量。指定

n_clusters的取值范围，每次运行k-means算法都能得到不同的聚类结果。一般而言，从融合的角度看，每次结果有所不同是件好事。否则，融合多次算法跟只运行一次算法效果相同（但是几个聚类结果之间差距较大也不能表明融合后效果就会好）。代码如下：

```
    def __init__(self, n_clusterings=10, cut_threshold=0.5, n_
clusters_range=(3, 10)):
        self.n_clusterings = n_clusterings
        self.cut_threshold = cut_threshold
        self.n_clusters_range = n_clusters_range
```

为EAC类定义fit函数。

```
    def fit(self, X, y=None):
```

接着，使用k-means算法进行低水平聚类，把每一次迭代得到的共协矩阵加起来。为了节省内存，我们使用生成器，仅在需要时创建共协矩阵。生成器每迭代一次，就在数据集上跑一个新的k-means实例，然后创建一个共协矩阵。再使用sum方法把得到的多个共协矩阵加起来。代码如下：

```
        C = sum((create_coassociation_matrix(self._single_
clustering(X))
                 for i in range(self.n_clusterings)))
```

跟前面一样，我们创建MST，删除低于阈值的边（注意使用相反数），找到连通分支。跟scikit-learn其他fit函数一样，我们都需要返回self，以便流水线的下一个步骤使用。代码如下：

```
        mst = minimum_spanning_tree(-C)
        mst.data[mst.data > -self.cut_threshold] = 0
        self.n_components, self.labels_ = connected_components(mst)
        return self
```

接着，编写通过一轮迭代进行聚类的函数。具体方法是，使用numpy的randint函数随机选取几个簇，用参数n_clusters_range指定随机选取的范围。然后使用k-means算法进行聚类和预测数据集属于哪个簇。最后返回由k-means计算得到的簇标签。代码如下：

```
    def _single_clustering(self, X):
```

```
        n_clusters = np.random.randint(*self.n_clusters_range)
        km = KMeans(n_clusters=n_clusters)
        return km.fit_predict(X)
```

像之前那样创建流水线，只不过在流水线最后一步，使用证据累积算法而不是之前的k-means算法实例，运行代码。代码如下。

```
pipeline = Pipeline([('feature_extraction', TfidfVectorizer(max_
df=0.4)),
                     ('clusterer', EAC())
                     ])
```

10.5　线上学习

有时，在开始学习之前，我们没有足够的数据用来进行训练。有时，数据量太大，内存装不下，或一时拿不到数据，或完成预测后，我们又拿到了新数据。以上情况就可以使用线上学习方法，在需要时及时训练模型。

10.5.1　线上学习简介

线上学习是指用新数据增量地改进模型。支持线上学习的算法可以先用一条或少量数据进行训练，随着更多新数据的添加，更新模型。相比之下，不支持线上学习的算法在开始训练之前需要一次性拿到所有数据。标准的k-means算法以及前几章的绝大部分算法都不支持线上学习。

线上学习算法在只有几条新数据的情况下就能做到部分更新已有模型。神经网络算法是支持线上学习的标准例子。随着一条新数据输入到神经网络后，网络中的权重根据学习速率进行更新，学习速率通常为一个很小的值，比如0.01。这表明一条新数据只能对模型带来很小的变动（希望是改进）。

神经网络还可以按照批模式来训练，每次只用一组数据进行训练。批模式，算法运行速度快，但是耗内存较多。

同理，我们可以用一个数据点或少量数据点来轻微更新k-means中的质心点。具体做法是，在k-means算法更新质心点这一步加入学习速率。我们假定这些新数据点是从总体中随机选取的，质心点应该朝它们在原来标准、离线的k-means算法中的位置移动。

线上学习与流式学习（streaming-based learning）有关，但有几个重要的不同点。线上学习能够重新评估先前创建模型时所用到的数据，而对于后者，所有数据都只使用一次。

10.5.2　实现

scikit-learn提供了MiniBatchKMeans算法，可以用它来实现线上学习功能。这个类实现了partial_fit函数，接收一组数据，更新模型。相反，调用fit()将会删除之前的训练结果，重新根据新数据进行训练。

因为MiniBatchKMeans聚类过程跟scikit-learn中其他聚类算法一致，所以创建和使用方法跟其他算法相同。

因此，我们可以使用TfIDFVectorizer从数据集中抽取特征，创建矩阵X。代码如下：

```
vec = TfidfVectorizer(max_df=0.4)
X = vec.fit_transform(documents)
```

再来导入MiniBatchKMeans，初始化一个实例。

```
from sklearn.cluster import MiniBatchKMeans
mbkm = MiniBatchKMeans(random_state=14, n_clusters=3)
```

接着，随机从X矩阵中选择数据，模拟来自外部的新数据。每次取一些数据，更新模型。

```
batch_size = 10
for iteration in range(int(X.shape[0] / batch_size)):
    start = batch_size * iteration
    end = batch_size * (iteration + 1)
    mbkm.partial_fit(X[start:end])
```

在实例上调用predict()方法，获得原始数据集的聚类结果。

```
labels  = mbkm.predict(X)
```

然而目前阶段，我们无法在流水线中使用TfIDFVectorizer，因为

它不是在线算法。为了解决这个问题，我们使用HashingVectorizer类，它巧妙地使用散列算法极大地降低了计算词袋模型所需的内存开销。我们不是记录特征名字，比如文档中的词，而是仅记录这些名字的散列值。这样，在我们查看数据集之前，就能知道所有的特征，因为我们已经拿到了所有特征的散列值。大约有2^{18}个散列值，数据量很大，但是使用稀疏矩阵，存储和计算都很容易，因为矩阵中很多项的值都为0。

写作本书时，sickit-learn的Pipeline类无法用于线上学习。不同的应用有着细微差别，这也就表明不存在万全之策，也就无法封装一个通用的流水线类。相反，我们可以创建自己的能够支持线上学习的流水线子类。我们这个类继承自scikit-learn的Pipeline类。

```
class PartialFitPipeline(Pipeline):
```

创建partial_fit函数，接收输入矩阵、可选的类别（这里用不到）。

```
    def partial_fit(self, X, y=None):
```

我们之前讲过，流水线由一系列转换步骤组成，前一步的输出为下一步的输入。我们把第一个输入设置为X矩阵，接着，用每一个转换器对输入进行转换。

```
        Xt = X
        for name, transform in self.steps[:-1]:
```

然后，转换已有的数据集，继续迭代，直到达到最后一步（我们这里为聚类算法）。

```
            Xt = transform.transform(Xt)
```

接着，在最后一步调用partial_fit函数，返回结果。

```
        return self.steps[-1][1].partial_fit(Xt, y=y)
```

我们现在就可以创建流水线，在线上学习过程使用

MiniBatchKMeans和HashingVectorizer。除了新类PartialFitPipeline和HashingVectorizer外，使用线上学习进行聚类分析，其整个过程跟本章其他部分大同小异，只不过每次使用的数据量少多了。

```
pipeline = PartialFitPipeline([('feature_extraction',
HashingVectorizer()),
                             ('clusterer', MiniBatchKMeans(random_
state=14, n_clusters=3))
                             ])
batch_size = 10
for iteration in range(int(len(documents) / batch_size)):
    start = batch_size * iteration
    end = batch_size * (iteration + 1)
    pipeline.partial_fit(documents[start:end])
labels = pipeline.predict(documents)
```

当然这种方法也有不足之处。比如，我们很难知道对于每个簇来说哪些词最为重要。如果想知道的话，这就要用到另外一种特征抽取工具CountVectorizer，先取到每个词的散列值，然后通过散列值而不是词本身查找词的重要性。这样做有点麻烦，并且抵消了我们使用HashingVectorizer节省下来的内存。并且，我们也无法使用之前用过的max_df参数，因为它需要知道特征是什么并一直统计它们。

此外，我们无法在线上学习中使用tf-idf权重。虽然我们可以估计并使用权重，但是也很麻烦。HashingVectorizer算法非常有用，是散列算法的精彩应用。

10.6　小结

本章研究了一种无监督学习方法——聚类。无监督学习用来探索数据而不是分类或预测。我们从reddit网站采集到的数据没有类别，所以无法对其进行分类。我们使用k-means算法把新闻报道分成组，以找到数据中隐藏的主题和趋势。

从reddit网站采集网址数据后，我们发现它们指向不同的网站，因此，需要研制一种适用范围广的网页内容抽取方法。抽取数据时，我们只寻找大块的文本，而没有使用成熟的机器学习方法。有一些有趣的机器学习方法可以改善文本抽取效果，详见附录部分本章的补充内容。我在附录中，针对每一章都给出了后续学习方向及实验效果改善方法，还给出了参考资料，并介绍了读者可能感兴趣的难度更大的应用。

本章还探讨了一种很直观的融合算法——证据累积算法。融合算法可用来处理结果之间的差异，尤其是你不知道怎么选取好的参数时（参数选取对于聚类来说就特别难）。

最后，我们介绍了线上学习。这是通向包括大数据在内的大型机器学习算法的入口，接下来两章会讲大数据。后两章的实验规模很大，在学习模型的同时，还要求学会管理数据。

下一章将从无监督学习再度回到分类。我们将研究以复杂神经网络为基础的分类方法——深度学习。

第 11 章　用深度学习方法为图像中的物体进行分类

第8章用到了最基础的神经网络。最近几年，该领域兴起的研究热潮为其带来了一系列长足的进步。如今，神经网络的研究创造出了一些最为先进和精确的分类算法，在很多领域展露锋芒。

计算机性能的提升使得训练更大、更复杂的神经网络成为可能，但该领域之所以能够取得进步，主要不是因为计算性能的提升，而是因为采用了新的算法和神经网络层。

本章研究如何确定图像中的物体，我们使用像素值作为神经网络的输入值，自动找到有用的像素组合，形成更高层级的特征，然后将其用于实际的分类。本章主要内容如下：

- 为图像中物体分类
- 不同类型的深度神经网络
- `Theano`、`Lasagne`和`Nolearn`：用于创建和训练神经网络的库
- 用GPU提升算法速度

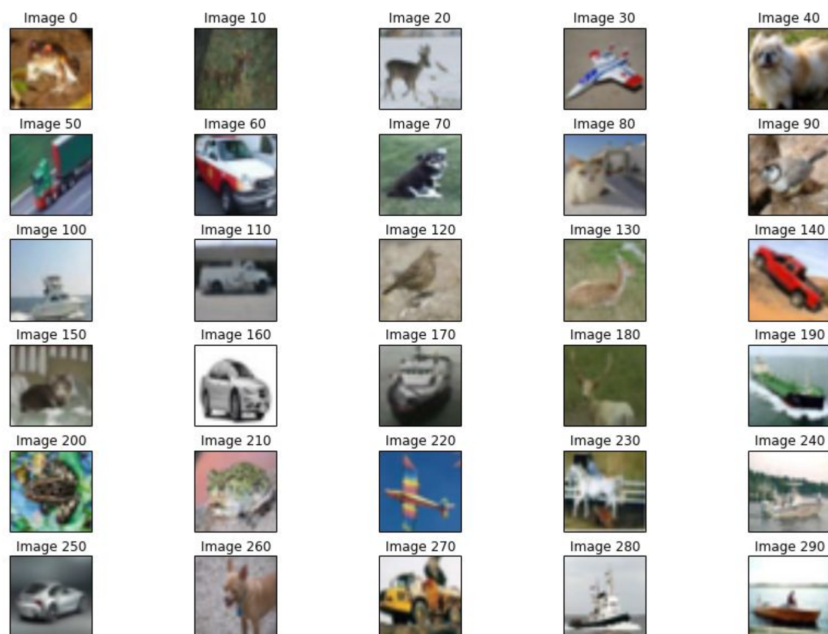
11.1　物体分类

计算机视觉逐渐成为科技领域的明日之星。未来五年，我们也许就能坐上自动驾驶汽车（甚至可能比这还要提前，如果流传的消息是可信的）。要让计算机来开车，它得能看清周围的环境：障碍物、其他车辆和天气状况等。

虽然计算机借助雷达等技术很容易就能检测到是否有障碍物，但这还不够，它还需要知道障碍物是什么。假如是动物，它们可能很快就会跑开，但如果是建筑物，那就麻烦了，计算机需要控制车辆躲开。

11.2　应用场景和目标

本章将构建一个系统，它接收图像，给出图像里面是什么物体。该系统就好比是自动驾驶汽车的视觉系统，它能发现道路及两侧的任何障碍物。我们使用如下形式的图像。



图像数据来自于著名的CIFAR-10数据集，它含有6万张32像素见方的图像，每个像素都有一个红—绿—蓝（RGB）值。这个数据集已经分为训练集和测试集两部分，我们在训练结束后才会用到测试集，这里先提一下。

> CIFAR10数据集的下载地址为http://www.cs.toronto.edu/~kriz/cifar.html。请下载Python版本，所有图像已经转换为numpy数组。

我们来看下这些图像数据长什么样，打开一个IPython Notebook笔记本文件，设置好文件名。我们开始只用第一批文件，到后面再增加数量使用全部数据集。

```
import os
data_folder = os.path.join(os.path.expanduser("~"), "Data", "cifar-10-batches-py")
batch1_filename = os.path.join(data_folder, "data_batch_1")
```

接着，创建一个函数，读取第一批图像文件数据。这些图像数据文件格式为pickle，pickle是Python用来保存对象的一个库。通常在pickle文件上调用pickle.load方法就能取得保存在里面的数据。但这里有个小问题：这些pickle文件是Python 2生成的，而我们要用Python3打开。所以在打开时需要把编码设置成latin（虽然我们是以字节模式打开）。

```
import pickle
# Bigfix thanks to: http://stackoverflow.com/questions/11305790/
pickle-incompatability-of-numpy-arrays-between-python-2-and-3
def unpickle(filename):
    with open(filename, 'rb') as fo:
        return pickle.load(fo, encoding='latin1')
```

使用上述函数加载数据集。

```
batch1 = unpickle(batch1_filename)
```

这一批图像文件读进来后为一个字典结构，包含numpy数组形式的图像数据、图像类别、文件名以及该批文件的简短说明（例如，training batch 1 of 5表示训练集共五批，这是第一批）。

以data作为键，从字典batch1中取到存储这一批图像数据的列表后，再用图像索引号就能取到其中一张图像的数据。

```
image_index = 100
image = batch1['data'][image_index]
```

每一张图像数据为一个numpy数组，数组共有3072项，每一项为0到255之间的一个值。每个值代表图像某一位置的红、绿或蓝色的颜色强度。

图像数据格式与matplotlib（绘制图像）所使用的有所不同，因此，用它来显示图像前，需要改变数组的形状，对矩阵进行转换。这种数据格式不会影响神经网络训练（我们定义网络时会考虑支持这种数据格式），但是要用matplotlib绘制图像，就需要对数据格式进行转换。

```
image = image.reshape((32,32, 3), order='F')
import numpy as np
```

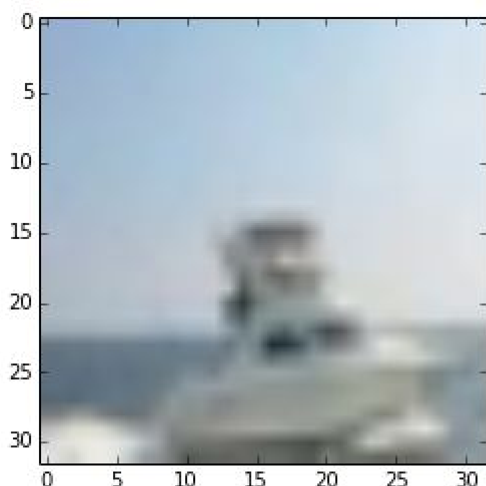
```
image = np.rot90(image, -1)
```

然后，就可以用matplotlib绘制图像。

```
%matplotlib inline

from matplotlib import pyplot as plt
plt.imshow(image)
```

这是张轮船图像。



图像分辨率很低——只有32像素见方。尽管如此，大部分人还是能看出图中是一艘船。我们能让计算机辨认出来吗？

你可以修改图像索引查看其他图像，了解下数据集的特点。

本章项目的目标是创建分类系统，预测像上面这样的图像的类别。

使用场景

计算机视觉技术应用范围很广。

在线地图网站，比如谷歌地图就使用计算机视觉技术，自动为街景中的人脸添加模糊效果，以保护被拍摄到的人们的隐私。除此之外，还有别的用途。

人脸识别技术在很多行业都有所应用。如今，照相机都能自动识别人脸，以提升拍摄质量（拍照时用户往往把相机聚焦到脸部）。人脸识别技术还可以用来识别照片中的人脸。Facebook就使用了这项技术，方便用户标记朋友。

我们前面也说过，自动驾驶汽车高度依赖计算机视觉来识别道路，躲避障碍物。计算机视觉是很多行业、应用都在努力解决的一个问题，不只是为了研制自动驾驶汽车，也不只是为了满足消费者的需求，采矿业等行业也在寻求攻克计算机视觉技术难关之路。

除此之外，其他行业也在使用计算机视觉技术，比如仓储业自动检测不合格产品就用到该项技术。

航天工业使用计算机视觉技术实现数据采集的自动化。这对于有效使用航天器来说至关重要，因为从地球上发送信号到火星上的探测车要花费很长时间，有时还会无法送达（例如，地球和火星不是面对面时）。人们跟太空探测器打交道越来越多，而从遥远的地球控制它们很不方便，增加这些设备的自主性就显得很有必要。

下图为美国国家航空和航天局（NASA）设计的火星车，它大量应用了计算机视觉技术。



11.3　深度神经网络

从理论角度看，第8章所使用的神经网络有几个显著的特点。例如，只需要一层隐含层来学习任何类型的映射（虽然隐含层可能规模很

大）。神经网络在20世纪七八十年代很火，但随后人们不再使用它们，尤其是跟支持向量机等其他分类算法相比，神经网络被冷落了。首要问题是，运行多个神经网络比其他算法对计算能力有更高的要求，超出了当时计算机的能力范围。

其次是训练神经网络所需工作量很大。虽然那时反向传播算法已经研制出来，处理大型神经网络仍存在问题，需要大量的训练才能确定权重。

这两个问题在近些年都得到了解决，神经网络迎来了又一个春天。计算能力比起30年前更容易获得，训练算法的改进使得在现有计算能力的情况下，能够顺利完成大型神经网络的训练。

11.3.1　直观感受

深度神经网络和第8章中的基本神经网络的差别在于规模大小。至少包含两层隐含层的神经网络被称为深度神经网络。实际工作中遇到的深度神经网络通常规模很大，每层神经元数量和层次都非常多。2000年年中的一些研究，关注的是层次数量多的深度神经网络，而更巧妙的算法能够减少实际所需要的层数。

神经网络接收很基础的特征作为输入——就计算机视觉而言，输入为简单的像素值。 神经网络算法把这些数据整合起来向网络中传输，在这个过程中，基本的特征组合成复杂的特征。有时，这些组合特征对我们来说没有什么含义，但是它们表示个体某些方面的特征，计算机依靠它们进行分类。

11.3.2　实现

由于深度神经网络规模很大，实现起来很有挑战。实现不好，运行时间会很长，甚至还会出现由于内存不足，最后无法运行的情况。

神经网络的基本实现可以从创建神经元类开始，多个神经元组成层类，每一层的神经元使用边这个类的一个实例连接到另一层神经元。这种基于类的实现，有助于显示网络的工作方式，但是对于创建大型网络，效率较低。

神经网络的核心其实就是一系列矩阵运算。两个网络之间连接的权重可以用矩阵来表示，其中行表示第一层的神经元，列表示第二层神经元（有时会用到该矩阵的转置矩阵）。矩阵的每一个元素表示位于不同层的两个神经元之间连接的权重。一个神经网络就可以用一组这样

的矩阵来表示。除了神经元外，每层增加一个偏置项，它是一个特殊的神经元，永远处于激活状态，并且跟下一层的每一个神经元都有连接。

对神经网络有了上述这般深入理解，我们就可以使用数学运算创建、训练和使用神经网络，比起前面用类来实现，效率要更高。就矩阵运算而言，有很多非常好用的库，其代码经过充分优化，借助它们可以高效完成矩阵运算。

第8章使用的PyBrain还实现了神经网络的卷积层。但是我们这里要用到的一些功能，它没有提供。对于规模更大，定制化程度更高的神经网络，我们需要更强大的库。因此，我们选用Lasagne和nolearn两个库。这两个库依赖于擅长处理数学表达式的Theano库。

本章首先通过用Lasagne实现一个基础的神经网络，来介绍相关概念。然后，使用nolearn库重新做第8章中的字母识别实验。最后，使用更为复杂的卷积神经网络对CIFAR数据集进行图像分类，为了提升性能我们使用GPU而不是CPU运行算法。

11.3.3　Theano简介

Theano是用来创建和运行数学表达式的工具。它用起来乍一看跟编写程序没有差别，但是在Theano中，我们定义函数要做什么而不是怎么做，这样Theano就能以最佳的方式对表达式进行求值，还可以进行延迟计算——只在需要时，对表达式求值，而不是定义时。

多数程序员不会每天都使用这种类型的编程范式，但是他们几乎天天接触使用这种编程范式的系统。关系型数据库，尤其是SQL类，就用到了叫作声明型范式（declarative paradigm）的概念。比如程序员定义了含有WHERE子句的SELECT查询语句。数据库解释查询语句，并根据一系列因素创建优化过的查询语句，数据库要考虑的因素有WHERE子句是否建立在主键之上、数据存储格式等。程序员决定他们要什么，系统决定怎么做。

> 你可以使用pip安装Theano：pip3 install Theano。

我们可以用Theano来定义函数，处理标量（scalars）、数组和矩阵及其他数学表达式。例如，我们可以创建计算直角三角形斜边长度的函数。

```
import theano
from theano import tensor as T
```

首先，定义两个输入a和b，它们为简单的数值类型，因此使用标量。

```
a = T.dscalar()
b = T.dscalar()
```

接着，定义输出c，它为由a和b构成的一个表达式。

```
c = T.sqrt(a ** 2 + b ** 2)
```

注意，上述c既不是函数，也不是一个数值——它是由a和b组成的表达式。a和b不是一个确切的值——这是一个代数式，最终结果不确定。为了计算这个表达式，我们来定义一个函数。

```
f = theano.function([a,b], c)
```

上述语句告诉Theano创建一个函数，这个函数接收a、b，返回经过计算得到的结果，也就是输出值c。例如，f(3， 4)，返回5。

虽然上述例子看起来可能不比Python普通函数强大多少，但是我们现在就在代码其他部分和后面的映射关系中使用我们的函数或数学表达式c。此外，虽然我们在创建函数之前，就定义了表达式，其实直到调用函数时，才会对表达式进行计算。

11.3.4 Lasagne简介

Theano库不是用来创建神经网络的，就好比numpy库不是用来执行机器学习任务的，而是被别的库使用，来出卖苦力的。Lasagne库是专门用来构建神经网络的，它使用Theano进行计算。

Lasagne实现了几种比较新的神经网络层和组成这些层的模块，具体介绍如下。

- 内置网络层（Network-in-network layers）：这些小神经网络比传统的神经网络层更容易解释。

- 删除层（Dropout layers）：训练过程随机删除神经元，防止产生神经网络常见的过拟合问题。

- 噪音层（Noise layers）：为神经元引入噪音，也是为了解决过拟合问题。

本章使用卷积层（convolution layers，层级结构模拟人类视觉工作原理）。卷积层使用少量相互连接的神经元，分析一部分输入值（比如我们这里的一张图像），便于神经网络实现对数据的标准转换，比如对图像数据的转换。视觉分析实验，就是用卷积层对图像进行转换 **1**。

> 1原文为translating the image。——译者注

比较而言，传统的神经网络内部连接十分紧密——一层的所有神经元全都连接到下一层所有神经元。

`lasagne.layers.Conv1DLayer`和 `lasagne.layers.Conv2DLayer classes`两个类实现了卷积网络。



写作本书时，`Lasagne`还没有正式版，资源没有上传到`pip`站点，因此无法用`pip`安装。但是可以从`github`安装。把`Lasagne`代码仓库克隆到本地新建的文件夹中：**`git clone https://github.com/Lasagne/Lasagne.git`**。进入到Lasagne文件夹，使用下面命令安装：

```
sudo python3 setup.py install
```

安装指南请见http://lasagne.readthedocs.org/en/latest/user/installation.html。

神经网络使用卷积层（一般而言，仅卷积神经网络包含该层）和池化层（pooling layer），池化层接收某个区域最大输出值，可以降低图像中的微小变动带来的噪音，减少（down-sample，降采样）信息量，这样后续各层所需工作量也会相应减少。

Lasagne还实现了池化层——比如lasagne.layers.MaxPool2DLayer类。再加上前面的卷积层，现在我们准备好了创建卷积神经网络所需的全部部件。

在Lasagne中创建神经网络比起只用Theano更加容易。我们通过实现一个简单的卷积神经网络，介绍相关规则。我们再次使用第1章所用到的Iris数据集，该数据集很适合用来测试新算法，即使是复杂的深度神经网络也没问题。

首先，打开一个新的笔记本文件。前面加载CIFAR数据集所使用的笔记本文件，先搁一边，后面还会用。

加载Iris数据集。

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data.astype(np.float32)
y_true = iris.target.astype(np.int32)
```

Lasagne对数据类型有特殊要求，因此，需要把类别值转换为int32类型（在原始数据集中用int64类型存储）。

把数据集分为训练集和测试集两部分。

```
from sklearn.cross_validation import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y_true, random_
state=14)
```

接着，分别创建卷积神经网络各层。我们的数据集有四个特征、三个类别，这样第一层和最后一层神经元数量就确定了，但是中间层多大呢？中间层大小不同，最终结果也会不同，尝试使用不同的值，看看结果会有怎样的变化。

首先，创建输入层，其神经元数量跟数据集特征数量相同。可以指定每一批输入的数量（设置为10），这样Lasagne可以在训练阶段做些优化。

```
import lasagne
input_layer = lasagne.layers.InputLayer(shape=(10, X.shape[1]))
```

接着，创建隐含层。该层从输入层接收输入（由第一个参数指定），该层共有12个神经元，使用非线性的sigmoid函数，我们在第8章曾经介绍过。

```
hidden_layer = lasagne.layers.DenseLayer(input_layer, num_units=12,
nonlinearity=lasagne.nonlinearities.sigmoid)
```

接下来，创建输出层，它接收来自隐含层的输入，输出层共有三个神经元（跟类别的数量一致），使用非线性的softmax函数，该函数主要用于神经网络的最后一层。

```
output_layer = lasagne.layers.DenseLayer(hidden_layer, num_units=3,
                                    nonlinearity=lasagne.
nonlinearities.softmax)
```

依照Lasagne的习惯用法，输出层为我们的神经网络。当我们输入一条数据到神经网络时，它查看输出层，向上回溯找到向输出层提供输入的那一层（第一个参数）。这个过程重复进行直到到达输入层，因为输入层没有上一层，所以就把要处理的数据交给输入层处理。输入层的激活函数把接收到的数据处理后输出给调用它的层（我们这里是隐含层），然后再一步步在网络中传播直到输出层。

为了训练刚创建的网络，我们需要定义几个Theano训练函数。在这之前，需要定义一个Theano表达式和函数。我们先来为神经网络的输入数据、输出结果和实际输出结果声明变量。

```
import theano.tensor as T
net_input = T.matrix('net_input')
net_output = output_layer.get_output(net_input)
true_output = T.ivector('true_output')
```

接着，定义损失函数，训练函数如何提升网络效果需要参考它的返回值——训练神经网络时，以最小化损失函数的返回值为前提。我们用类别交叉熵（categorical cross entropy）表示损失，这是一种衡量分类数据（categorical data）分类效果好坏的标准。损失函数表示的是网络的期望输出和实际输出两者之间的差距。

```
loss = T.mean(T.nnet.categorical_crossentropy(net_output,
  true_output))
```

接着，定义修改网络权重的函数。我们需要获取到网络的所有参数，创建调整权重的函数（使用Lasagne提供的工具），使得损失降到最小。

```
all_params = lasagne.layers.get_all_params(output_layer)
updates = lasagne.updates.sgd(loss, all_params, learning_rate=0.1)
```

最后，创建两个Theano函数，先是训练网络，然后获取网络的输出，以用于后续测试。

```
import theano
train = theano.function([net_input, true_output], loss,
updates=updates)
get_output = theano.function([net_input], net_output)
```

然后调用训练函数，在训练集上进行一轮迭代，接收训练数据，预测类别，与给定类别作比较，更新特征权重，以最小化损失。然后再进行1000次迭代，逐渐改进神经网络。

```
for n in range(1000):
    train(X_train, y_train)
```

接着，对输出计算F值，以评估分类效果。首先获取到所有输出。

```
y_output = get_output(X_test)
```

注意，get_output为Theano函数，是我们从神经网络中获取到的，因此，上述代码不用再把神经网络作为参数传进去。

y_output为输出层每个神经元的激励作用（activation）大小。找出激励作用最高的神经元，就能得到预测结果。

```
import numpy as np
y_pred = np.argmax(y_output, axis=1)
```

数组y_pred为预测结果数组，跟我们在前面分类任务中用到的相同。然后，我们就可以计算F1值。

```
from sklearn.metrics import f1_score
print(f1_score(y_test, y_pred))
```

结果异常完美——1.0！这表明我们的算法为测试集中所有数据正确分类，这个结果太棒了（虽然数据集有点小）。

从上面我们可以看到，仅用Lasagne库也能创建、训练一个神经网络，但是有点麻烦。我们可以使用nolearn库，对上述过程做进一步封装，使它很容易就能与scikit-learn接口对得上。

11.3.5 用nolearn实现神经网络

nolearn对Lasagne进行了封装，虽然比起用Lasagne创建神经网络，使用nolearn无法对一些细节进行调整，但是代码可读性更强，也更容易管理。

nolearn实现了几种常见的复杂程度很高的神经网络，应该能够满足你的需求。如果你想拥有更多的控制权，可以回头继续使用Lasagne，但是创建和训练过程需要你多花点心思。

为了快速了解nolearn，我们再次来实现第8章中识别图像中字母的实验。我们重建在第8章所创建的密集神经网络（dense neural network）。再次在笔记本文件输入创建数据集的代码。这些代码的具体含义，请参考第8章。

```
import numpy as np
from PIL import Image, ImageDraw, ImageFont
from skimage.transform import resize
from skimage import transform as tf
from skimage.measure import label, regionprops
from sklearn.utils import check_random_state
from sklearn.preprocessing import OneHotEncoder
from sklearn.cross_validation import train_test_split

def create_captcha(text, shear=0, size=(100, 24)):
    im = Image.new("L", size, "black")
    draw = ImageDraw.Draw(im)
    font = ImageFont.truetype(r"Coval.otf", 22)
    draw.text((2, 2), text, fill=1, font=font)
    image = np.array(im)
    affine_tf = tf.AffineTransform(shear=shear)
    image = tf.warp(image, affine_tf)
    return image / image.max()

def segment_image(image):
```

```
    labeled_image = label(image > 0)
    subimages = []
    for region in regionprops(labeled_image):
        start_x, start_y, end_x, end_y = region.bbox
        subimages.append(image[start_x:end_x,start_y:end_y])
    if len(subimages) == 0:
        return [image,]
    return subimages

random_state = check_random_state(14)
letters = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
shear_values = np.arange(0, 0.5, 0.05)

def generate_sample(random_state=None):
    random_state = check_random_state(random_state)
    letter = random_state.choice(letters)
    shear = random_state.choice(shear_values)
    return create_captcha(letter, shear=shear, size=(20, 20)),
letters.index(letter)
dataset, targets = zip(*(generate_sample(random_state) for i in
range(3000)))
dataset = np.array(dataset, dtype='float')
targets =  np.array(targets)

onehot = OneHotEncoder()
y = onehot.fit_transform(targets.reshape(targets.shape[0],1))
y = y.todense().astype(np.float32)

dataset = np.array([resize(segment_image(sample)[0], (20, 20)) for
sample in dataset])
X = dataset.reshape((dataset.shape[0], dataset.shape[1] * dataset.
shape[2]))
X = X / X.max()
X = X.astype(np.float32)

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, train_size=0.9, random_state=14)
```

神经网络由一系列的层组成。在nolearn中实现神经网络，只需定义它由Lasagne哪几种类型的层组成就行，跟PyBrain做法大同小异。第8章所创建的神经网络使用的是全连通密集层。nolearn有相应的实现，这表明我们可以用nolearn实现的各层来复制第8章神经网络的基本结构。首先，创建由输入层、密集隐含层和密集输出层组成的层级结构。

```
from lasagne import layers
layers=[
        ('input', layers.InputLayer),
        ('hidden', layers.DenseLayer),
```

```
        ('output', layers.DenseLayer),
        ]
```

导入以下所需模块，用到时再解释它们的作用。

```
from lasagne import updates
from nolearn.lasagne import NeuralNet
from lasagne.nonlinearities import sigmoid, softmax
```

接着，定义神经网络，它与scikit-learn估计器兼容。

```
net1 = NeuralNet(layers=layers,
```

注意上面代码没加右半边括号——我们有意为之。输入神经网络的参数，先从每层的大小开始吧。

```
    input_shape=X.shape,
    hidden_num_units=100,
    output_num_units=26,
```

上述参数跟每一层相对应。换句话说，input_shape参数就会去查找名称为input的输入层，这跟在流水线中设置参数的工作原理很相似。

接着，定义非线性函数。跟之前一样，隐含层使用sigmoid函数，输出层使用softmax函数。

```
    hidden_nonlinearity=sigmoid,
    output_nonlinearity=softmax,
```

接下来，指定偏置神经元，它们位于隐含层，一直处于激活状态。偏置神经元对于网络的训练很重要，神经元激活后，可以对问题做更有针对性的训练，以消除训练中的偏差。举个简单的例子，如果预测结果总是偏差4，我们可以使用-4偏置值抵消偏差。我们设置的偏置神经元就能起到这样的作用，训练得到的权重决定了偏置值的大小。

偏置项为一组权重值，它所包含的元素数量应与它所连接的层的大小相同。

```
    hidden_b=np.zeros((100,), dtype=np.float32),
```

接下来，定义神经网络的训练方式。我们在第8章中用到的训练方法，nolearn没有与之完全相同的，因为nolearn没有削减权重的方法。然而，它的确有冲量（momentum），我们这里使用一个高学习速率和低冲量值。

```
    update=updates.momentum,
    update_learning_rate=0.9,
    update_momentum=0.1,
```

接下来，我们把分类问题定义为回归问题。这看起来有点奇怪，因为这是分类任务。然而，别忘了输出结果是一组数值，在训练阶段把它当成回归问题进行优化比起分类问题效果更好。

```
    regression=True,
```

最后，最大训练步数（epoch）设置为1000，这样既能保证训练效果，又不至于花太长时间（对于我们数据集很合适，但其他数据集训练步数会有所差异）。

```
    max_epochs=1000,
```

终于可以把创建神经网络这个函数的右半边括号加上了。

```
)
```

接着，在训练集上训练网络。

```
net1.fit(X_train, y_train)
```

现在，我们来评估训练得到的网络。首先，获取神经网络的输出，跟Iris分类一样，我们需要使用argmax方法找到输出值中最大的一项，然后找到它所对应的类别。

```
y_pred = net1.predict(X_test)
y_pred = y_pred.argmax(axis=1)
```

```
assert len(y_pred) == len(X_test)
if len(y_test.shape) > 1:
    y_test = y_test.argmax(axis=1)
print(f1_score(y_test, y_pred))
```

结果同样振奋人心——在我的计算机上，再次全部预测正确。然而，你可能会得到不同的结果，因为nolearn库具有一定的随机性，眼下无法直接控制。

11.4 GPU优化

神经网络体积可能会增长得很大，这意味着需要更多内存；然而，使用稀疏矩阵这样高效的存储方式把整个神经网络装进内存一般不会遇到问题。

神经网络体积增长带来的主要问题是计算时间增加。此外，对于有些数据集，神经网络需要更多的训练步数才能较好地拟合数据集。本章所创建的神经网络在我性能还不错的计算机上，每完成一步训练需要八分多钟，而我们需要运行数十步甚至成百上千步训练。体积更大的神经网络，每一步训练需要几个小时，而为了取得最佳效果，可能需要成千上万步训练。

多长时间才能完成训练啊！

好在神经网络最核心的计算类型主要为浮点型运算。此外，由于神经网络训练过程主要为矩阵操作，大量运算都可以并行处理。这些因素表明用GPU进行计算，能提升训练速度。

11.4.1 什么时候使用GPU进行计算

GPU的设计初衷是用于图像渲染。这些图像用矩阵及由矩阵组成的数学方程式来表示，它们被转换为我们在屏幕上看到的像素。这个过程涉及大量并行计算。如今的CPU很可能是多核（你的计算机可能有2、4，甚至16个核——或更多！），GPU却拥有专门用于图像处理的成千上万个小核。

CPU每个核单独工作速度往往更快，访问计算机内存等操作效率更高，因而适合处理序列化任务。说实话，让CPU出苦力更容易，因为几乎所有的机器学习库默认使用CPU，如果要用GPU做并行计算，需要很多额外工作，但是它所带来的好处也是显而易见的。

有些任务，包含大量数字运算，但计算量都不大，可以同时处理，这样的任务最好交由GPU来处理。很多机器学习任务都具有类似的特点，因此使用GPU处理能提升算法运行速度。

让代码在GPU上跑起来可不简单，中间可能会遇到各种各样的挫折，具体与GPU类型、配置方式、操作系统相关，有时你可能还需要修改计算机底层设置。

使用GPU运算，主要方法有以下三种。

- 第一种，使用现有计算机。了解你的计算机型号，查找GPU、操作系统相关的软件和驱动程序，寻找相关教程，软件、教程是否能用与操作系统相关。不过，现在使用GPU，比前几年容易多了，有更多更好的软件和驱动程序来启动GPU计算。

- 第二种，选定计算机系统，找到关于如何设置系统的文档，购买一台装有这种系统的新计算机，买的系统很好用，但是可能相当昂贵——如今的计算机，GPU是价格最高的部件之一。尤其是你对性能有更高要求的话——你需要一个价格不菲的好GPU。

- 第三种，使用专门为GPU计算而配置的虚拟机。例如，Markus Beissinger在亚马逊的AWS上搭建了一个这样的系统，该系统不是免费的，你得付费才能使用它提供的服务，但是比起购买一台新计算机要便宜得多。具体费用由你所在的区域、使用的系统类型和资源消耗数量来决定，你可能每小时花不到1美元，往往比这还要少得多。如果你使用AWS的竞价型实例（spot instance），每小时只需几美分（然而你需要单独开发在竞价型实例上运行的代码）。

如果你无法承担虚拟机开销，我建议你使用上面提到的第一种方法，即使用现有的计算机。你还可以试试看能不能从隔段时间就要换装备的家人或朋友那里找到二手GPU（爱玩游戏的朋友很可能有哦！）。

11.4.2　用GPU运行代码

我们这里将使用上述第三种方法，在Markus Beissinger系统的基础上创建一个虚拟机，该虚拟机运行在亚马逊的EC2云平台上。除了EC2，还有很多其他Web服务可供使用，但方法可能跟这里有所不同。我会讲下如何在亚马逊云平台上搭建虚拟机。

如果你打算使用自己的计算机，并且已经配置好，可以用GPU进行计算了，请跳过该节。

> 关于如何进行设置的更多信息，请见http://markus.com/installtheano-on-aws/，也许还提供在其他计算机上启用GPU计算的方法。

下面，看下如何在AWS上创建虚拟机。首先，访问AWS的登录界面。

https://console.aws.amazon.com/console/home?region=us-east-1

用你的AWS账号登录。如果没有，得先创建一个才能继续下面的操作。

接着，访问EC2云平台的控制台：https://console.aws.amazon.com/ec2/v2/home?region=us-east-1。

点击Launch Instance，从右上方的下拉菜单中选择N. California作为目的地。

点击Community AMIs，搜索ami-b141a2f5，这就是Markus Beissinger创建的系统。然后，点击Select。下一屏中的机器类型选择g2.2xlarge，点击Review and Launch。在下一屏，点击Launch。

这个时候，AWS开始向你收费，所以使用完毕后请关机。从EC2云平台选择你刚创建的云主机，先停止运行。机器处于停机状态，不需要支付费用。

系统会给出如何连接虚拟机的相关信息。如果你之前没有使用过AWS，你可能需要创建密钥以安全连接虚拟机。创建时，需要指定密钥的名称，下载.pem格式的密钥文件，将其保存到一个安全地方——一旦丢失，你将无法登录自己的虚拟机！

点击Connect，了解如何使用密钥文件连接虚拟机。你很可能是用如下ssh命令登录。

```
ssh -i <certificante_name>.pem ubuntu@<server_ip_address>
```

11.5 环境搭建

连接到虚拟主机后，你可以安装最新的Lasagne和nolearn库。

首先，使用git命令克隆Lasagne代码仓库，本章前面用过。

```
git clone https://github.com/Lasagne/Lasagne.git
```

在虚拟机上编译这个库，需要用到Python 3的setuptools，我们可以用Ubuntu常用的应用和包管理命令apt-get来安装；我们还需要numpy的开发版。

在虚拟机的命令行运行下述命令。

```
sudo apt-get install python3-pip python3-numpy-dev
```

接着，安装Lasagne。首先，切换到源代码所在的目录，然后运行setup.py完成安装。

```
cd Lasagne
sudo python3 setup.py install
```

> 我们已经安装了Lasagne，还需要安装nolearn，为了简单起见，把它俩都作为系统包来安装。考虑到可移植性，建议你使用virtualenv安装这些包，这样你可以在同一台虚拟机上使用不同版本的Python和第三方包，把代码移植到其他机器上也更容易。更多信息请见http://docs.pythonguide.org/en/latest/dev/virtualenvs/。

Lasagne编译好后，我们就可以安装nolearn。切换到你自己的主目录，照下面的步骤来，跟前面安装Lasagne的步骤相同，只不过要注意这次克隆的是nolearn。

```
cd ~/
git clone https://github.com/dnouri/nolearn.git
cd nolearn
sudo python3 setup.py install
```

我们的环境差不多搭建好了。还需要安装scikit-learn、scikit-image库，这两个库都可以用pip3安装，这些库所依赖的

scipy和matplotlib也没有安装。建议使用apt-get来安装scipy和matplotlib，因为使用pip3可能会遇到比较棘手的问题。

```
sudo apt-get install python3-scipy python3-matplotlib
sudo pip3 install scikit-learn scikit-image
```

接着，需要把代码搬到虚拟机上，方法有很多，最简单的莫过于复制粘贴。

首先，打开前面用过的笔记本文件（在你本地的计算机上，不是在亚马逊的虚拟机上）。笔记本文件顶部有一个菜单。点击File，然后再点击Download as，选择Python，将其保存到本地计算机上。该步骤将笔记本文件保存为可以从命令行运行的.py文件。

打开.py文件（有些操作系统，你可能需要右键单击该文件，从弹出的菜单中选择用文本编辑器打开）。文件打开后，选择所有的内容，将其复制到剪贴板。

回到亚马逊的虚拟主机，切换到主目录，用nano文本编辑器打开一个新文件。

```
cd ~/
nano chapter11script.py
```

nano为命令行文本编辑器，上述命令将在命令行打开新文件。

文件打开后，把剪贴板的内容粘贴到里面。在有些系统中，你可能需要借助ssh程序的文件操作命令来完成文件上传，而不是按快捷键Ctrl＋V粘贴。

在nano中，按下Ctrl＋O保存文件，再按Ctrl＋X退出nano。

你还需要字体文件。最简单的方法是从原地址再下载一遍，具体步骤如下。

```
wget http://openfontlibrary.org/assets/downloads/bretan/680bc56bbeeca9535
3ede363a3744fdf/bretan.zip
sudo apt-get install unzip
unzip -p bretan.zip Coval.otf > Coval.otf
```

上述代码用unzip命令解压Coval.otf文件（压缩包里有大量的zip文件，我们用不到）。

在虚拟机的命令行里输入以下命令运行程序。

```
python3 chapter11script.py
```

程序将会像在IPython Notebook笔记本里那样运行，把结果输出到命令行。

结果应该跟之前一致，但是神经网络的训练和测试比之前更快。程序其他方面可能没那么快——生成验证码数据集部分没有使用GPU，因此速度不会提升。

> 你可能希望关闭亚马逊虚拟主机来省点钱，本章最后运行大实验的相关程序时会再次用到虚拟机，我们接下来先在本地计算机上进行开发。

11.6　应用

回到本地计算机，打开本章创建的笔记本文件——我们用来加载CIFAR数据集的。接下来的大实验将使用CIFAR数据集，创建深度卷积神经网络，然后在虚拟机上使用GPU来运行。

11.6.1　获取数据

首先，用CIFAR图像创建数据集。跟之前不同的是，保留像素结构——像素的行列号。先把所有批次的图像文件名存储到列表中。

```
import numpy as np
batches = []
for i in range(1, 6):
    batch_filename = os.path.join(data_folder, "data_batch_{}".
format(i))
    batches.append(unpickle(batch1_filename))
    break
```

上面最后一行的break语句是用来测试代码——这会极大地减少训练数据，便于你迅速了解代码是否能正常运行。测试过代码能正常工作后，我会提示你删掉这一行。

接着，把每批次的图像文件依次添加到数据集里。我们用到了NumPy的vstack方法，你可以把它看作是往数组末尾追加一行数据。

```
X = np.vstack([batch['data'] for batch in batches])
```

然后，把像素值归一化，并将其强制转换为32位浮点型数据（这是使用GPU进行计算的虚拟机唯一支持的数据类型）。

```
X = np.array(X) / X.max()
X = X.astype(np.float32)
```

类别数据处理方法相同，只不过我们使用hstack，它为数组末尾追加一列数据。然后使用OneHotEncoder把它转换为只含有一位有效编码的数组。

```
from sklearn.preprocessing import OneHotEncoder
y = np.hstack(batch['labels'] for batch in batches).flatten()
y = OneHotEncoder().fit_transform(y.reshape(y.shape[0],1)).todense()
y = y.astype(np.float32)
```

接下来，把数据集切分为训练集和测试集。

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
size=0.2)
```

调整数组形状以保留原始图像的数据结构。图像原本是32像素见方，每个像素由三个值组成（分别表示红、绿、蓝的颜色值）。

```
X_train = X_train.reshape(-1, 3, 32, 32)
X_test = X_test.reshape(-1, 3, 32, 32)
```

我们现在准备好了训练集、测试集以及目标类别，可以创建分类器了。

11.6.2　创建神经网络

我们使用nolearn库创建神经网络，步骤跟我们上面重做第8章实验时所用到的一致。

首先，创建神经网络的各层。

```
from lasagne import layers
layers=[
        ('input', layers.InputLayer),
        ('conv1', layers.Conv2DLayer),
        ('pool1', layers.MaxPool2DLayer),
        ('conv2', layers.Conv2DLayer),
        ('pool2', layers.MaxPool2DLayer),
        ('conv3', layers.Conv2DLayer),
        ('pool3', layers.MaxPool2DLayer),
        ('hidden4', layers.DenseLayer),
        ('hidden5', layers.DenseLayer),
        ('output', layers.DenseLayer),
        ]
```

最后三层使用密集层，但是前面使用三组卷积层和池化层。此外，我们（必须）以输入层开始。这样一共有10层。输入数据跟数据集同型，而不只跟输入层的神经元数量相等，输入层和输出层大小仍由数据集来定，跟前面讲过的相同。

开始创建神经网络（注意下面第二行代码没有结束，先不要加右半边括号）。

```
from nolearn.lasagne import NeuralNet
nnet = NeuralNet(layers=layers,
```

指定输入数据的形状，跟数据集形状一致（每个像素由三个值组成，图像32像素见方）。第一个值None表示nolearn每次使用默认数量的图像数据进行训练——它将立即用这些数据进行训练，降低算法运行时间。设置为None，避免硬编码，算法使用起来更灵活。

```
                input_shape=(None, 3, 32, 32),
```

如果你想每次使用不同的数据量进行训练，就需要创建BatchIterator 实例。感兴趣的读者可参考如下文件：https://github.com/dnouri/nolearn/blob/master/nolearn/lasagne.py，了解NeuralNet类中batch_iterator_train和batch_iterator_test参数是如何设置的。

接着，设置卷积层的大小。没有严格的规则可遵守，但是我发现一开始用下面这些值就不错。

```
                conv1_num_filters=32,
                conv1_filter_size=(3, 3),
                conv2_num_filters=64,
                conv2_filter_size=(2, 2),
                conv3_num_filters=128,
                conv3_filter_size=(2, 2),
```

filter_size参数规定卷积层用于查看图像窗口的大小。此外，还要设置池化层的大小。

```
                pool1_ds=(2,2),
                pool2_ds=(2,2),
                pool3_ds=(2,2),
```

然后，设置两层隐含层（倒数第三层和倒数第二层）、输出层的大小，输出层大小跟数据集类别数量相等。

```
                hidden4_num_units=500,
                hidden5_num_units=500,
                output_num_units=10,
```

最后一层需要设置非线性函数，还是用softmax。

```
                output_nonlinearity=softmax,
```

还要设置学习速率和冲量。据经验来看，随着数据量的增加，学习速率应该下降。

```
                update_learning_rate=0.01,
                update_momentum=0.9,
```

跟之前一样，把regression参数设置为True，训练步数设置为很小的值3，因为我们创建的这个神经网络运行时间相对较长。神经网络确定能正常运行之后，可以尝试增加训练步数以得到更好的模型，但是可能要花一两天时间（甚至更长！）来训练它。

```
```

```
                regression=True,
                max_epochs=3,
```

最后一个参数verbose设置为1，这样每步训练都会输出结果，以便于我们了解模型的训练进度以及它是否在运行。此外，还能输出每一步训练所花的时间，这个值变化不大，因此用每步训练所花时间乘以剩余训练步数就能算出总共还需要多长时间才能完成训练。

```
                verbose=1)
```

11.6.3　组装起来

现在就可以在训练集上训练我们刚创建的神经网络。

```
nnet.fit(X_train, y_train)
```

这可真是要费点时间了，即使我们只使用了部分训练数据，并且还限制了训练步数。一旦代码运行结束，测试方法跟之前一样。

```
from sklearn.metrics import f1_score
y_pred = nnet.predict(X_test)
print(f1_score(y_test.argmax(axis=1), y_pred.argmax(axis=1)))
```

结果很糟糕——本应如此！我们没有经过充分训练——只进行了三轮迭代，且只使用了1/5的数据。

首先，回过头去删除创建数据集代码中的break语句（在遍历每批数据的for循环中）。这样就能使用所有数据而不只是部分数据进行训练。

接着，在神经网络的定义中，把训练步数改为100。

现在，把代码粘贴到虚拟机上。跟之前一样，点击File | Download as，选择Python，把代码保存为.py文件。启动、连接到虚拟机，像之前把代码那样粘贴过去（我把这里的.py文件命名为chapter11cifar.py——如果你使用其他名字，请记得在下面代码中做相应改动）。

接下来，我们需要把数据集捣腾到虚拟机上。最简单的方法是在虚拟

机的命令行输入以下命令。

```
wget http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
```

下载数据集，新建Data文件夹，把数据文件解压到Data文件夹中。

```
mkdir Data
tar -zxf cifar-10-python.tar.gz -C Data
```

最后，使用下面命令运行实验代码。

```
python3 chapter11cifar.py
```

你首先注意到的将是速度大幅提升。在我的本地计算机上，每步训练需要100秒以上。在启用GPU的虚拟机上，每步训练只需16秒！在我本地计算机上进行100步训练，需要近3个小时，而虚拟机只需要26分钟。

速度上的大幅提升使得我们可以快速测试不同模型的效果。对于机器学习算法调试来说，一个算法的计算复杂性问题不大，只要几秒钟、几分钟，多则几个小时就可以运行完。因此，只有一个模型，训练时间长点短点不会有太大问题——特别是大多数机器学习算法都能很快给出预测结果，这也正是为什么大多数情况都使用一个模型。

然而，要调试多个参数时，你就需要训练成千上万个模型，各模型之间参数差异很小——速度提升问题就变得很重要。

26分钟后，完成100步训练，得到最终的输出结果。

```
0.8497
```

结果还不错！我们可以增加训练步数，进一步改善效果或者也可以尝试修改参数值，增加隐含层节点、卷积层的数量，或者多使用一层密集层。也可以尝试用Lasagne其他类型的层，虽然一般来讲，卷积层更适合处理视觉问题。

11.7　小结

本章讲解了深度神经网络，为了处理计算机视觉问题，着重讲解卷积网络。我们使用Lasagne和nolearn库来构建神经网络，很多数据处理工作交由Theano来做。用nolearn提供的工具构建神经网络相对比较容易。

卷积神经网络专门用来处理计算机视觉问题，所以最后得到的分类结果很精确也不足为奇。最终结果表明，随着算法和计算能力的提升，计算机视觉的应用前景也更为广阔。

我们在虚拟机上使用GPU计算，大幅提升训练速度，在虚拟机上运行的速度几乎是我本地计算机的10倍。如果某些算法需要额外的计算能力，可以考虑使用性价比很高的云主机（一般每小时不到1美元）——只要记得用完后即时关机就好！

本章所研究的卷积网络算法相当复杂。卷积网络训练时间长，有很多参数需要训练。数据集看似不小，其实还称不上大数据，甚至不用稀疏矩阵，就能把这些数据全部加载到内存。下章所讲的算法更加简单，但是数据集却特别大，无法装进内存，这是大数据最显著的一个特点，采矿业和社交网络等行业所产生的数据都具有这个特点。

第 12 章　大数据处理

当今社会数据量呈指数级增长，这些数据来自于用户行为监测系统、分布式系统、网络分析、传感器等。移动互联网的迅速发展又带来移动数据的增长，此外，下一个大潮流——物联网（Internet of Things，IoT）又会进一步提升数据增长速度。

挖掘大数据需要有新思路。运行时间很长的复杂算法要么改进要么抛弃，而能够处理更多数据，复杂程度相对简单的算法正变得更加流行起来。例如，支持向量机是很好的分类器，但是它的一些变种很难在大型数据集上使用。相反，逻辑回归等相对简单的算法处理大数据集更容易。

本章主要内容如下:

- 大数据的挑战及其应用
- MapReduce范式
- Hadoop MapReduce
- 在亚马逊平台上运行MapReduce程序的Python包mrjob

12.1　大数据

大数据具有哪些特点呢？它的大多数支持者认为有以下4个特点，简称4V。

(1) 海量（Volume）：我们产生和存储的数据量加速增长，未来势头看起来会更猛。现在常用的硬盘容量单位为GB，再过几年就会变为EB1。同时网络吞吐量也会增长。信噪比不容乐观，重要信息很可能被海量重要程度低的信息湮没。

> 1EB（exabyte），艾字节，1EB＝1024PB，1PB＝1024TB，1TB＝1024GB。1EB约等于10亿GB。——译者注

(2) 高速（Velocity）：数据量增长的同时，数据处理速度在加快。就拿现如今的汽车举例子，每辆车装有成百上千个传感器，这些传感器

不停地向汽车内置的计算机传送数据，要完成操纵汽车的任务，数据的分析处理速度需要达到次秒级水准。这不仅要从大量数据中寻找答案，还得速度快。

(3) 多样（Variety）：数据集有多种形式，各列清晰定义的规整数据集只占其中很少一部分。想想社交网站的一条消息吧，它可能包含文本、照片、提及其他用户2、喜欢数、评论数、视频、地理位置信息等。简单地忽略掉这些难以整合进模型的数据势必会导致信息丢失。

> 2指“@”加用户名这种形式。——译者注

(4) 准确（Veracity）：随着数据量的增加，很难确定采集到的数据是否正确——是否过时、充满噪音，有没有包含异常数据，或者总体来说是否有用等。倘若人们无法对数据的准确性进行有效验证，要信赖数据很难。人们从外部得到的数据集被不停地整合到内部数据集中，使得数据的准确性这个问题更加突出。

从以上四点（也有人认为是五点3）就能看出大数据不仅仅是大量的数据，处理大数据对工程能力要求很高——更不用说对它们进行分析了。很多“蛇油商人”4只关心怎样夸大大数据的应用，却避而不谈大数据工程的挑战以及潜在分析工作的难度。

> 3有人认为大数据的另一个特点是价值（Value），经过挖掘得到的结论有实际应用价值。——译者注

> 4蛇油商人（snake oil salesman）跟“江湖郎中”有相同的贬义内涵。19世纪，国人赴美修建北太平洋铁路，带去从南方水蛇体内提炼的蛇油，以祛风湿，缓筋骨之痛，颇受当地人欢迎。后美国商人Clark Stanley等改从北美响尾蛇提炼蛇油，批量生产，但效果较差，甚至出现产品中压根不含蛇油的造假现象。该词遂被用来指打着不容置疑的旗号，推销假冒伪劣谋求私利的骗子。——译者注

我们前面用过的这些算法都是把数据集加载到内存后再进行处理。这对于提升计算速度很有好处，因为处理已经在内存中的数据比起先从硬盘加载到内存后再处理要快得多。此外，我们可以对内存中的数据进行多次迭代，改善模型。

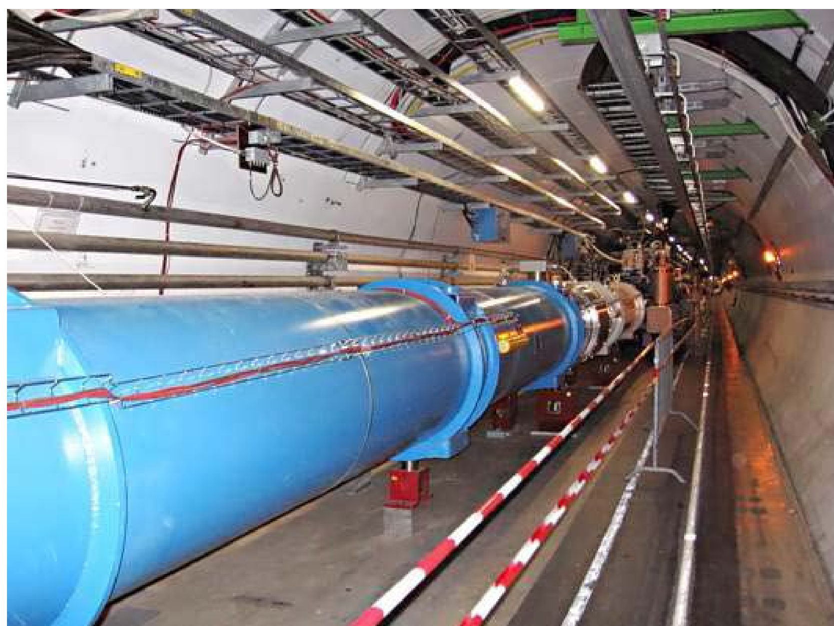
对于大数据，我们就无法将其加载到内存中。从很多方面来讲，可以用此来判断一个问题能不能称得上大数据问题——如果数据可以加载到你计算机内存中，那就算不上大数据。

12.2　大数据应用场景和目标

企业和个人都离不开大数据。

所有以大数据为基础的系统中，人们与之打交道最多的当属谷歌这样的搜索引擎。对于搜索引擎来说，要在零点几秒的时间内对几十亿甚至更多的网站完成一次搜索，基础的文本查询肯定满足不了需求，单是存储这么多网站的文本内容就是个大难题。要完成搜索引擎中的查询任务，就需要专门创建新数据结构和使用新数据挖掘方法。

很多其他科学实验也都用到了大数据技术，例如大型强子对撞机（Large Hadron Collider）。该机器长达27公里，装有1.5亿个传感器，用于监测每秒几亿个强子的对撞情况，对撞机局部照片请见下图。对撞实验产生的数据量十分惊人，每天过滤后还有25PB（假如不过滤，每年将产生1.5亿PB数据）。这些数据隐藏着宇宙的奥秘，对其进行分析能加深我们对宇宙的认识，但是工程量很大，分析起来也相当困难。



政府部门对大数据的重视程度也在逐渐增加，以充分了解人口、商业活动和国家其他各方面的情况。跟踪几千万乃至几亿人口及成百上千亿的交易量（商贸或医疗开支），很多政府机构都需要依靠大数据分析技术才能完成。

全球很多政府尤其关注交通管理，他们使用数以千万计的传感器判断哪些道路堵塞最为严重，预测新修道路对交通状况的影响。

大型零售商使用大数据改善客户体验，降低支出。这包括预测客户需求，保证供货量适度，向客户推销他们可能喜欢的产品，跟踪交易数据，寻找购物潮流和消费模式，发现潜在欺诈行为。

还有很多大公司用大数据提高公司经营管理的自动化程度，改善产品和服务质量。他们通过大数据分析预测行业风向，跟踪外部竞争者。他们还把大数据用到员工管理上——跟踪员工，发现其离职倾向以及时干预。

信息安全部门通过监测网络流量，结合大数据技术，寻找大型网络的恶意软件感染，这包括寻找奇怪的流量模式，恶意软件传播迹象和其他反常现象。高级持续性威胁（Advanced Persistent Threats，APTs）攻击也很令人抓狂，攻击者把恶意代码藏在大型网络中，长期从网络中窃取信息或从事其他破坏活动。寻找APTs往往需要检查很多台计算机进行取证，费时费力，仅靠人工很难完成，这时就可以使用大数据技术实现自动化检查、取证。

大数据方兴未艾，正在越来越多的行业和应用中大显身手。

12.3　MapReduce

大数据挖掘和计算有几个主要概念，其中最为流行的就是MapReduce模型，它可用于任意大数据集的一般性计算任务。

谷歌出于并行计算的需要，最先提出了MapReduce模型。它还引入了容错和可伸缩特性。MapReduce最初的研究发表于2004年，打那时起，成千上万的项目、实现和应用都用到了这一概念。

虽然MapReduce跟之前很多概念有相似之处，但这不影响它成为大数据分析领域的核心概念。

12.3.1　直观理解

MapReduce主要分为两步：映射（Map）和规约（Reduce）。函数式程序设计中有把函数映射到列表和规约结果的概念，MapReduce以这两个概念为基础。为了解释映射和规约，我们举个例子，编写代码遍历一串列表，计算所有列表中数字之和。

MapReduce范式还包括排序（shuffle）和合并（combine）两步，后面会讲。

首先，在映射这一步，接收一个函数，用这个函数处理列表的各元素，返回跟之前列表长度相等的列表，新列表的元素为函数的返回结果。

打开一个新的IPython Notebook笔记本文件，创建一列表，列表中各元素为包含几个数字的列表。

```
a = [[1,2,1], [3,2], [4,9,1,0,2]]
```

接着，建立sum函数和a之间的映射关系。这一步会用sum函数处理列表a的每一个元素。

```
sums = map(sum, a)
```

上述sums为生成器（在我们调用它之前，不会进行计算），上面这行代码大体上等价于：

```
sums = []
for sublist in a:
    results = sum(sublist)
    sums.append(results)
```

规约步骤要稍微复杂些，需要对返回结果的每一个元素应用一个函数，从初始值开始，对初始值和第一个值应用指定函数，得到返回结果，然后再对所得到的结果和下一个值应用指定函数，以此类推。

我们来创建一个函数，接收两个数字作为参数，返回两个数字的和。

```
def add(a, b):
  return a + b
```

然后进行规约。规约函数形式为reduce(function, sequence, initial)，参数分别表示用来进行规约的函数的名字、列表和初始值，函数即是对序列的每一步所应用的函数。在第一步，第一个值为初始值，而不是列表的第一个元素。

```
from functools import reduce
print(reduce(add, sums, 0))
```

结果为25，它是sums列表中每个元素的和，也就是原来大列表a中每个二级列表各元素的和。

上面代码等价于如下代码。

```
initial = 0
current_result = initial
for element in sums:
    current_result = add(current_result, element)
```

我们这个小例子，代码很简短，但真正的好处是可以使用分布式计算。假如，二级列表的数量为100万，每个二级列表包含100万个元素，对于这么大的计算任务，我们可以交由多台计算机完成。

为了实现分布式计算，我们可以在映射这一步把各个二级列表及函数说明分发到不同的计算机上。计算完成后，各计算机把结果返回主计算机（master）。

然后master把结果发送给另一台计算机做规约。我们的例子有100万个二级列表，因此可以将100万个任务交给不同的计算机处理（计算机完成映射操作后，可再次用于规约）。返回结果为包含100万数字的列表，然后就可以进行求和运算。

这样做的好处是，尽管原始数据集有1万亿数字，但实际计算过程，哪台计算机都不需要存储超过100万个数字。

12.3.2 单词统计示例

MapReduce的实现比起单纯使用映射和规约两步要复杂些。这两步都用键来调用，便于数据的区分和值的跟踪。

映射函数接收一键值对，返回键值对列表。接收和输出的键不一定要彼此相关。例如，统计单词数量的MapReduce程序，输入的键可能是文档编号，而输出的键却可以是单词。输入值可能是文档的文本内容，而输出值为每个单词的词频。

```
from collections import defaultdict
def map_word_count(document_id, document):
```

```
```

首先，计算每个单词词频。在这个简化过的例子中，我们先根据空格把文档转换成单词列表，当然有更好的解决方法。

```
    counts = defaultdict(int)
    for word in document.split():
        counts[word] += 1
```

遍历每个单词，统计次数，注意用到了`yield`语句。用MapReduce的术语来说，单词为键，单词出现次数为值。

```
    for word in counts:
        yield (word, counts[word])
```

用单词做键，我们就可以进行shuffle操作，把每个键所有值聚集到一起。

```
def shuffle_words(results):
```

首先，把每个单词的统计结果放到一个列表中。

```
    records = defaultdict(list)
```

遍历映射函数返回的所有结果。

```
    for results in results_generators:
        for word, count in results:
            records[word].append(count)
```

接着，遍历每个单词，创建生成器，它能生成（yield）单词和该单词在各文档出现次数的列表这两项。

```
    for word in records:
        yield (word, records[word])
```

最后一步是规约，接收一键值对（值为列表），输出另外一键值对。

我们这里，键为单词，输入的列表为shuffle后得到的作为键的单词在不同文档出现次数的列表，输出键值对中的值这一项为单词（键）在所有文档的出现次数之和。

```
def reduce_counts(word, list_of_counts):
    return (word, sum(list_of_counts))
```

我们使用scikit-learn提供的来自20个新闻组的语料看下上述代码的实际效果。

```
from sklearn.datasets import fetch_20newsgroups
dataset = fetch_20newsgroups(subset='train')
documents = dataset.data
```

然后执行映射操作，这里用enumerate函数自动为文档生成编号。编号这里用处不大，但是在其他应用中很重要。

```
map_results = map(map_word_count, enumerate(documents))
```

上述结果只是生成器，而不是实际的统计结果。也就说，它是能输出键值对（单词、出现次数）的生成器。

接着，对生成器进行shuffle操作，根据单词出现次数进行排序。

```
shuffle_results = shuffle_words(map_results)
```

上述例子本质上来说是一个MapReduce任务；然而由于只使用单线程，我们无法从MapReduce数据格式中获得任何好处。下节，我们将使用开源的MapReduce架构Hadoop，实现分布式计算，提升性能。

12.3.3　Hadoop MapReduce

Hadoop是Apache基金会所提供的一组包括MapReduce在内的开源工具。人们实际使用的MapReduce架构也多是它。Hadoop项目由Apache基金会管理（他们也负责维护著名的Web服务器软件Apache）。

Hadoop生态系统很复杂，有大量的工具。我们将要用到的主要组件

为Hadoop MapReduce。Hadoop其他处理大数据的工具有如下几种。

- Hadoop分布式文件系统（Hadoop Distributed File System，HDFS）：该文件系统可以将文件保存到多台计算机上，以防范硬件故障，提高带宽。
- YARN：用于调度应用和管理计算机集群。
- Pig：用于MapReduce的高级语言。Hadoop MapReduce用Java语言实现，Pig对Java实现做进一步封装，支持用其他语言来编写程序——包括Python。
- Hive：用于管理数据仓库和进行查询。
- HBase：对谷歌分布式数据库BigTable的一种实现。

这些工具都是用来解决包括分析过程在内的大数据实验所能遇到的各种问题。

还有其他一些没有使用Hadoop工具集实现的MapReduce框架，也有一些具有相似目标的项目。此外，很多云平台都提供以MapReduce为基础的系统。

12.4　应用

我们来建立一个根据博主用词习惯判断博主性别的应用。我们用MapReduce方法训练朴素贝叶斯分类器。最后用模型做预测时，不需要MapReduce，虽然我们可以用映射这个步骤——也就是对列表中的每一篇文档运行预测模型。这是用MapReduce进行数据挖掘常用的映射操作，规约步骤只用来调整预测结果列表，以便把结果和原文档对应起来。

我们将使用亚马逊云平台运行应用，以利用它们的计算资源。

12.4.1　获取数据

我们使用2004年8月份从http://blogger.com网站采集的60多万篇博客文章作为数据集，这些文章共有1.4亿多个单词，每篇文章都标注了博主的年龄、性别、行业（工作），有趣的是还有星座。研究人员曾进行过部分验证，确保同一篇博客从头到尾均由一名博主所写（虽

然也不是很确定）。每篇博客还给出了发表时间，这个数据集的内容真是挺丰富的。

访问http://u.cs.biu.ac.il/~koppel/BlogCorpus.htm网站，点击Download Corpus。下载完成后，把数据集文件解压到你的本地计算机数据文件夹Data里。

数据集中，一个博主的所有博客放到同一个文件里，文件名包含博主的相关信息。例如，其中一个文件名如下：

```
1005545.male.25.Engineering.Sagittarius.xml
```

文件名用点号分隔，主要包括以下信息。

- 博客编号：标识博客唯一性的数值。
- 性别：该部分不是male（男性）就是female（女性）。数据集只包含这两种情况。
- 年龄：给出确切的年龄，但有时会特意使用年龄段。年龄段（包含起止年龄）有13~17、23~27和33~48。这样做是便于把难以确定博主年龄的博客粗略归到某一年龄段中，因为很难把18岁写的博客和19岁所写的区分开来，此外，当你使用这些数据时，博主实际年龄可能比早先填写的又长了几岁，因此需要做相应调整。5
- 行业：包括科学、工程、艺术、房地产在内的40种行业的其中一种，若博主没有填写该项，使用indUnk6来表示。
- 星座：12星座之一。

> 5作者给我举了个例子，“去年用户发表博客时为18岁，今年就19岁了，我需要在模型中做相应调整。”——译者注

> 6industry unknown的简写。——译者注

所有的这些数据都是用户自己提供的，这就表明其中有错误或不一致现象，但是绝大多数是可靠的——如果用户为了保护个人隐私起见，不想透露某方面的信息，他们可以选择不填，这一点网站是允许的。

每个文件的格式类似于XML，包含<Blog>标签和一系列<post>标签。每个<post>标签前都有一个<date>标签。虽然我们可以把它当作XML来处理，但是按行处理更容易，因为它不是真正规范的XML文件，并且还有些错误（大部分是编码错误）。我们可以使用循环结构遍历文件中的每一行，以读取博客内容。

指定一个文件名，我们来实际看下。

```
import os
filename = os.path.join(os.path.expanduser("~"), "Data", "blogs",
"1005545.male.25.Engineering.Sagittarius.xml")
```

首先，创建用于存储每篇博客的列表。

```
all_posts = []
```

然后，打开要读取的文件。

```
with open(filename) as inf:
```

设置标识是否在博客中的标记。找到博客开始标签<post>后，将标记的值设置为True，表示找到了博客的开始位置。找到关闭标签</post>后，将标记的值设置为False。

```
    post_start = False
```

创建用于存储博客当前行内容的列表。

```
    post = []
```

遍历文件的每一行，删除该行前后的空格。

```
    for line in inf:
        line = line.strip()
```

照前面所说，找到博客的开始、关闭标签后，更改标记post_start的值。

```
        if line == "<post>":
            post_start = True
        elif line == "</post>":
            post_start = False
```

找到关闭标签`</post>`后，记录刚找到的这一篇博客的所有内容。创建新的post列表。下面两行代码的缩进字符数与前一行代码相同。

```
            all_posts.append("\n".join(post))
            post = []
```

最后，如果该行不是博客结尾，也就是还在博客里面，把这一行加到当前博客post列表的最后。

```
        elif post_start:
            post.append(line)
```

如果没有在博客中，忽略这一行。

可以像下面这样获取每篇博客的文本内容。

```
print(all_posts[0])
```

我们还可以计算出一位博主共发表了多少博客。

```
print(len(all_posts))
```

12.4.2　朴素贝叶斯预测

我们来实现朴素贝叶斯算法（从技术上讲，这是一个简化的版本，复杂实现通常有很多特征）处理博客博主性别分类问题。

1. mrjob包

把用mrjob创建的MapReduce任务推送到亚马逊云平台上很容易。mrjob听上去有点像奇先生7系列儿童故事书的画蛇添足之作，实则表示映射规约任务（Map Reduce Job）。这个包很有用，但是写作本书时，对Python 3的支持还不成熟，对我们后面要讲到的亚马逊EMR服

务支持也不够好。

> 7英国儿童文学作家罗杰·哈格里维斯（Roger Hargreaves）为儿童创作的《奇先生妙小姐》系列图书中的人物形象，比如有荒唐先生、傲慢先生、邋遢先生等。——译者注



你可以使用下面命令安装mrjob的Python 2版本：

```
sudo pip2 install mrjob
```

注意使用`pip2`而不是`pip3`。

本质上讲，mrjob提供了大部分MapReduce任务所需的标准功能。最令人惊异的特性是，你可以编写同一套代码，既能在没有安装Hadoop的本地计算机上进行测试，测试完后，还可以直接把代码提交到亚马逊的EMR或其他Hadoop服务器上。

这样，测试起代码来很容易，虽然它也不可能神奇地把大问题变成小问题——注意任何本地测试，只是使用一小部分而不是全部数据。

2. 抽取博客内容

首先创建一个MapReduce程序，从每位博主的博客文件中抽取所有博客，分别存储。因为最终要预测博主的性别，我们还需要抽取性别信息，跟博客一起存储。

我们这回没法用笔记本文件，因此使用Python IDE进行开发。如果你没有装Python IDE（比如PyCharm），你可以使用文本编辑器。建议你使用具有代码高亮功能的IDE。

如果你找不到好用的IDE，你可以在笔记本文件中编写代码，然后点击（或选择）File | Download As | Python把代码保存成.py文件，然后再运行。第11章提到过这种方法。

因为需要获取环境变量，我们会用到`os`，单词拆分时会用到正则表达式，一并导入`re`。

```
import os
import re
```

```

```

接着导入MRJob类，我们的MapReduce任务继承自它。

```
from mrjob.job import MRJob
```

然后创建MRJob的子类。

```
class ExtractPosts(MRJob):
```

我们使用跟之前类似的循环从文件中抽取博客内容。我们即将定义的映射函数将处理每一行，跟踪不同博客的任务要在映射函数外完成。因此，在函数外而不是在函数内部声明post_start和post两个变量。

```
    post_start = False
    post = []
```

然后创建映射函数——从文件中取一行作为输入，最后生成一篇博客的所有内容。每一行都来自同一任务所在处理的文件。这样，我们就可以使用上面创建的变量保存当前博客的内容。

```
    def mapper(self, key, line):
```

开始采集博客内容之前，我们还要获取到博主的性别。虽然通常我们不会把文件名作为MapReduce任务的一部分，但是这里确实要用到。当前的文件名称以环境变量的形式存储，用下面代码就能获取到。

```
        filename = os.environ["map_input_file"]
```

用点号切分文件名，获取性别（第二个字符串）。

```
        gender = filename.split(".")[1]
```

删除一行开头和结尾处的空格（博客中空格很多），然后查找文件开头和结尾。

```
        line = line.strip()
        if line == "<post>":
            self.post_start = True
        elif line == "</post>":
            self.post_start = False
```

之前我们把博客保存到列表里，而这次我们用yield，便于mrjob跟踪输出：博主性别和博客内容，这样博主性别和博客就能一一对应起来。函数余下部分跟前面用到的循环中的代码一致。

```
            yield gender, repr("\n".join(self.post))
            self.post = []
        elif self.post_start:
            self.post.append(line)
```

最后，在函数和类的外面，编写下面语句，以便从命令行运行代码时执行MapReduce任务。

```
if __name__ == '__main__':
    ExtractPosts.run()
```

现在，可以在命令行输入下面的shell命令运行MapReduce任务。注意使用Python 2而不是Python 3。

```
python extract_posts.py <your_data_folder>/blogs/51* --output-
  dir=<your_data_folder>/blogposts -no-output
```

第一个参数<your_data_folder>/blogs/51*（注意把<your_data_folder>替换为你的数据文件夹的全路径）指定所使用的数据集（所有以51开始的文件，只有11个）。接着指定输出文件夹，即用来保存博客内容的文件夹。最后指定不要在命令行输出内容。否则，程序运行期间，输出的数据将显示在命令行——对我们来说没多大用处，还会降低程序运行速度。

运行上述代码，每篇博客内容很快就会被抽取出来并保存到指定的输出文件夹。在本地计算机上，只使用一个线程，所以速度提升有限，但是我们知道代码确实可以运行了。

我们看下输出文件夹，里面有有一系列文件，文件的每一行为博主的性别及一篇博客的内容。

3. 训练朴素贝叶斯分类器

既然已经抽取了博客内容，接下来就可以用它们训练朴素贝叶斯模型。根据直觉，我们可以这样做，分别统计女性博主和男性博主使用每个单词的概率。对博客进行分类时，我们分别计算博客8博主为女性的概率和博主为男性的概率，选取概率较大的，作为最终类别。

> 8一篇博客，其博主为女性的概率有多大？计算方法为，从语料中统计到该博客中每个单词出现在女博主所写博客中的概率，然后求出这些概率的积。其中会涉及到数值下溢、博客中某一单词在语料中没有出现等问题，这些问题需要特殊处理。下文也会讲到。——译者注

我们来编写代码，输出所有文件中每个单词及女博主和男博主使用该词的频率。输出文件如下所示：

```
"'ailleurs"  {"female": 0.003205128205128205}
"'air"  {"female": 0.003205128205128205}
"'an"  {"male": 0.0030581039755351682, "female": 0.004273504273504274}
"'angoisse"  {"female": 0.003205128205128205}
"'apprendra"  {"male": 0.0013047113868622459, "female":
0.0014172668603481887}
"'attendent"  {"female": 0.00641025641025641}
"'autistic"  {"male": 0.002150537634408602}
"'auto"  {"female": 0.003205128205128205}
"'avais"  {"female": 0.00641025641025641}
"'avait"  {"female": 0.004273504273504274}
"'behind"  {"male": 0.0024390243902439024}
"'bout"  {"female": 0.002034152292059272}
```

输出结果每一行的第一个值为单词，第二值为一个字典，字典的键为性别，字典的值为给定性别使用该词的频率。

用Python IDE或文本编辑器新建一个文件，再次要用到`os`和`re`，除此之外还要用到NumPy和MRJob。因为要对字典进行排序，所以还用到itemgetter。

```
import os
import re
import numpy as np
from mrjob.job import MRJob
from operator import itemgetter
```

我们还需要用MRStep管理MapReduce中的每一步操作。前面的

MapReduce任务只有由映射函数和规约函数组成的一个步骤。我们当前这个任务分三步：映射、规约、再次映射和规约。是不是感觉跟前几章我们用到的流水线一样，前一步的输出将作为下一步的输入。

```
from mrjob.step import MRStep
```

接着创建用于匹配单词的正则表达式，并对其进行编译，我们用它来查找单词的边界。这种类型的正则表达式比起前面用过的split方法更加强大，如果你需要更加精确的单词切分工具，建议你使用第6章用到的NLTK库。

```
word_search_re = re.compile(r"[\w']+")
```

创建一个新类，用于训练朴素贝叶斯分类器。

```
class NaiveBayesTrainer(MRJob):
```

定义MapReduce任务的各个步骤，一共分为两步。第一步抽取单词出现的概率。第二步，比较一个单词在男女博主所写博客出现的概率，选择概率较大的性别作为分类结果，写入输出文件。在上述每一步（MRStep）中，定义映射和规约函数，它们是NaiveBayesTrainer类里面的方法（后续会编写这两个函数）。

```
    def steps(self):
        return [
            MRStep(mapper=self.extract_words_mapping,
                   reducer=self.reducer_count_words),
            MRStep(reducer=self.compare_words_reducer),
            ]
```

第一个函数是第一步中的映射函数。这个函数的目标是接收一条博客数据，获取里面的所有单词，因为我们想要得到单词的频率，所以每个单词返回1 / len(all_words)，便于后面加总求词频。这样计算并不精确——我们需要根据文档数量做归一化处理。但由于数据集中两个类别的文档数相同，因此不做归一化处理对最终结果影响也很小。

输出博主的性别，后面会用到。

```
def extract_words_mapping(self, key, value):
    tokens = value.split()
    gender = eval(tokens[0])
    blog_post = eval(" ".join(tokens[1:]))
    all_words = word_search_re.findall(blog_post)
    all_words = [word.lower() for word in all_words]
        all_words = word_search_re.findall(blog_post)
        all_words = [word.lower() for word in all_words]
        for word in all_words:
            yield (gender, word), 1. / len(all_words)
```

上面代码使用eval函数简化对每条博客数据的处理，直接把字符串转换为Python列表，但不建议这么做。相反，应该使用JSON等格式存储数据，然后用相应包进行解析。如果在访问数据集的代码中使用eval语句，攻击者可借此插入恶意代码，并在你的服务器上运行。

在第一步的规约函数中，汇总每个性别使用每个单词的频率。我们还把键改为单词，而不是单词和性别的组合，因为在最后训练得到的模型中，我们要根据单词进行查询（虽然还要输出性别以供后面使用）。

```
def reducer_count_words(self, key, frequencies):
    s = sum(frequencies)
    gender, word = key
    yield word, (gender, s)
```

最后一步不需要映射函数，因此我们就没有添加它。数据将作为一致性映射（identity mapper）类型直接传入到规约函数中，而规约函数将会把每个单词在所有文章中出现频率按照性别汇集到一起，输出单词及频率字典。

这正是朴素贝叶斯分类器所需要的信息。

```
def compare_words_reducer(self, word, values):
    per_gender = {}
    for value in values:
        gender, s = value
        per_gender[gender] = s
    yield word, per_gender
```

最后，添加以下代码，以便直接运行代码时，训练朴素贝叶斯模型。

```
if __name__ == '__main__':
    NaiveBayesTrainer.run()
```

运行上述代码，它的输入即是前面博客抽取代码的输出（实际上可以把它们作为前后相连的步骤，放到同一个MapReduce任务中）。

```
python nb_train.py <your_data_folder>/blogposts/
  --output-dir=<your_data_folder>/models/
nooutput
```

上述代码运行结束后，该MapReduce任务的输出结果保存到位于输出文件夹中的一个文件里，输出结果为运行朴素贝叶斯分类器所需的概率信息。

4. 组装起来

我们现在就可以使用这些概率来运行贝叶斯分类器。我们将在笔记本文件里编写代码，可以再次使用Python 3喽！

首先，查看上一个MapReduce任务所生成的模型文件夹。如果输出文件多于一个，在命令行切换到模型文件夹下，使用下面命令把它们追加到model.txt后面。

```
cat * > model.txt
```

运行上述命令后，记得把下面用到的模型所在的文件名改为model.txt。

回到笔记本文件，导入接下来会用到的几个包。

```
import os
import re
import numpy as np
from collections import defaultdict
from operator import itemgetter
```

重新定义用于查找单词的正则表达式——在实际应用中，不要忘记这点，注意训练和测试时应该使用相同的正则表达式来抽取单词。

```
word_search_re = re.compile(r"[\w']+")
```

```

```

接着，声明从指定文件名中加载模型的函数。

```
def load_model(model_filename):
```

模型的参数是一个元素为字典的字典，各元素的键为单词，值（内部字典）为由每个性别及其概率组成的键值对。我们使用defaultdict，如果键不存在的话，将返回0。

```
    model = defaultdict(lambda: defaultdict(float))
```

打开模型所在文件，解析每一行。

```
    with open(model_filename) as inf:
        for line in inf:
```

用空格作为分隔符，将模型的每一行切分成两部分。第一部分为单词自身，第二部分为概率字典。对于每一部分，使用eval函数获得实际的值，它们之前是使用repr函数存储的。

```
            word, values = line.split(maxsplit=1)
            word = eval(word)
            values = eval(values)
```

在模型中，为单词和概率字典建立起映射关系。

```
            model[word] = values
    return model
```

接着，加载实际的模型。你可能需要修改模型的文件名——它存储在上一个MapReduce任务的输出文件夹中。

```
model_filename = os.path.join(os.path.expanduser("~"), "models",
"part-00000")
model = load_model(model_filename)
```

举例来说，我们可以像下面这样查看不同性别使用单词i（执行MapReduce任务时，所有单词中的字母全部转换为小写）的情况。

```
model["i"]["male"], model["i"]["female"]
```

接着，创建使用模型做预测的函数。这里我们不再使用scikit-learn接口，只是创建一个简单的函数。这个函数接收模型和一篇文档作为参数，返回对博主性别的预测结果，函数声明如下：

```
def nb_predict(model, document):
```

先来创建一个字典，键值分别为每个性别及概率9。

9不同性别用词有自己的偏好。此处的概率指的是，待预测文档每一个单词由给定性别所使用的概率的乘积。——译者注

```
    probabilities = defaultdict(lambda : 1)
```

从文档中抽取每一个单词。

```
    words = word_search_re.findall(document)
```

遍历每一个单词，找出数据集中每个性别使用该单词的概率。

```
    for word in set(words):
        probabilities["male"] += np.log(model[word].get("male", 1e-15))
        probabilities["female"] += np.log(model[word].get("female", 1e-15)
```

根据概率对性别进行排序，以概率最高的性别作为预测结果返回。

```
    most_likely_genders = sorted(probabilities.items(),
key=itemgetter(1), reverse=True)
    return most_likely_genders[0][0]
```

注意，我们使用np.log计算概率。朴素贝叶斯模型中，概率值往往很小。把这些很小的数连乘起来会导致数值下溢，由于计算机精度不够，最终结果将为0。在我们这里，就会出现博主为男性或女性的概

率都为0的情况，预测结果自然不正确。

为了解决数值下溢问题，我们使用对数概率。对于两个数a和b，`log(a,b)`等于`log(a) + log(b)`。数值很小的数取对数，得到一个负数，但相对还比较大。例如`log(0.00001)`约等于-11.5。这表明与其冒着数值下溢的风险，把多个概率连乘，还不如使用对数概率求和，然后比较两类的和（和越大，可能性越高）。

使用对数概率有个问题就是无法处理0值（虽然，把多个为0的概率连乘也不能解决这个问题）。这是因为`log(0)`没有定义。有些朴素贝叶斯算法，为每个特征的计数都加1以避免出现这个问题，当然还有别的解决方法。加一平滑是一种很简单的数据平滑方法。我们这里如果某个单词给定性别没有人用过，就给它赋一个很小的概率值。

再回到我们的预测函数，我们从数据集中复制一篇博客来做下测试。

```
new_post = """ Every day should be a half day.  Took the afternoon
off to hit the dentist, and while I was out I managed to get my oil
changed, too.  Remember that business with my car dealership this
winter?  Well, consider this the epilogue.  The friendly fellas at the
Valvoline Instant Oil Change on Snelling were nice enough to notice
that my dipstick was broken, and the metal piece was too far down in
its little dipstick tube to pull out.  Looks like I'm going to need a
magnet.   Damn you, Kline Nissan, daaaaaaammmnnn yooouuuu....   Today
I let my boss know that I've submitted my Corps application.  The news
has been greeted by everyone in the company with a level of enthusiasm
that really floors me.     The back deck has finally been cleared off
by the construction company working on the place.  This company, for
anyone who's interested, consists mainly of one guy who spends his
days cursing at his crew of Spanish-speaking laborers.  Construction
of my deck began around the time Nixon was getting out of office.
"""
```

用下面代码进行分类。

```
nb_predict(model, new_post)
```

分类结果为男性，分类正确。当然，千万不要只用一条数据进行测试。我们只使用文件名以51打头的文档来训练模型，所以不要指望有太高的正确率。

当务之急是用更多的文档来训练模型。我们使用所有文件名以6或7打头的文档做测试，在剩余文档上进行训练。

使用命令行，切换到博客文档所在的文件夹（cd <your_data_folder>），复制文档到新文件夹中。

创建训练集文件夹。

```
mkdir blogs_train
```

然后，创建测试集文件夹。

```
mkdir blogs_test
```

把所有文件名以6或7打头的文档从训练集文件夹移动到测试文件夹。

```
cp blogs/6* blogs_train/
cp blogs/7* blogs_train/
```

我们需要返回训练集中所有博客的抽取结果。然而，计算量很大，最好使用云平台而不是本地计算机处理这个任务。因此，我们接下来使用亚马逊云平台解析博客内容。

跟之前一样，在命令行运行下面代码。唯一的不同点是训练集所在的文件夹路径。运行代码前，删除用于存储抽取结果和模型文件夹中的所有文件。

```
python extract_posts.py ~/Data/blogs_train --output-dir=/home/bob/
Data/blogposts -no-output
python nb_train.py ~/Data/blogposts/ --output-dir=/home/bob/models/
nooutput
```

上述代码运行时间较长。

我们使用测试集中的所有博客进行测试。我们使用 extract_posts.py MapReduce任务抽取博客内容，但是记得把抽取结果保存到另外一个文件夹中。

```
python extract_posts.py ~/Data/blogs_test --output-dir=/home/bob/Data/
blogposts_testing -no-output
```

回到笔记本文件，获取到输出的所有测试文档的路径。

```
testing_folder = os.path.join(os.path.expanduser("~"), "Data",
"blogposts_testing")
testing_filenames = []
for filename in os.listdir(testing_folder):
    testing_filenames.append(os.path.join(testing_folder, filename))
```

对于上面输出的每一篇测试文档，我们抽取博主性别和博客内容，然后调用预测函数进行分类。因为文档数量很多，我们不想占用太多内存，所以使用生成器来处理。生成器会给出实际性别和预测结果。

```
def nb_predict_many(model, input_filename):
    with open(input_filename) as inf:
        # remove leading and trailing whitespace
        for line in inf:
            tokens = line.split()
            actual_gender = eval(tokens[0])
            blog_post = eval(" ".join(tokens[1:]))
            yield actual_gender, nb_predict(model, blog_post)
```

然后我们记录测试集中所有数据的预测结果和实际性别。预测结果不是男性就是女性。为了使用scikit-learn的f1_socre函数，我们需要把预测结果转换为0或1，如果结果为男性，我们记录0，反之为1。我们使用布尔测试，判断性别是否为女性，然后使用NumPy把布尔值转换为整型（int）。

```
y_true = []
y_pred = []
for actual_gender, predicted_gender in nb_predict_many(model, testing_
filenames[0]):
    y_true.append(actual_gender == "female")
    y_pred.append(predicted_gender == "female")
y_true = np.array(y_true, dtype='int')
y_pred = np.array(y_pred, dtype='int')
```

现在，我们使用scikit-learn中的F1值评估分类结果。

```
from sklearn.metrics import f1_score
print("f1={:.4f}".format(f1_score(y_true, y_pred, pos_label=None)))
```

结果为0.78，还算可以。我们使用更多的数据也许能改善分类效果。

但是，我们需要使用更强大的云平台才能处理更多数据。

5. 在亚马逊EMR云平台上训练模型

我们接下来使用亚马逊的EMR（Elastic Map Reduce）云平台完成博客抽取、解析和模型创建任务。

首先，需要在亚马逊的云存储（storage cloud）创建存储段（bucket）。具体做法是用浏览器打开亚马逊S3控制界面http://console.aws.amazon.com/s3，点击Create Bucket。记住存储段的名字，稍后会用到。

右键点击新建的存储段，选择`Properties`。然后，修改权限，将其设置为对所有人开放。一般来说，这样做不安全，学完这一章后，请及时修改权限。

左键点击存储段，打开它，点击Create Folder，把新文件夹命名为blogs_train。我们将把训练集数据上传到该文件夹，以便在云端处理。

我们在本地计算机上使用亚马逊的AWS CLI命令行工具操作亚马逊云平台。

使用以下命令安装该工具。

```
sudo pip2 install awscli
```

按照网址中给出的说明，为CLI工具设置安全证书：http://docs.aws.amazon.com/cli/latest/userguide/cli-chap-getting-set-up.html。

接下来需要上传数据到新建的存储段。首先，我们创建训练集，它包括所有除去文件名以6或7打头的文件。复制文件有很多种更为优雅的方式，但是考虑到平台兼容性，就不做推荐。相反，我们使用最牢靠的做法，复制所有的原始文件到训练集文件夹，把文件名以6、7打头的从训练集中删除。

```
cp -R ~/Data/blogs ~/Data/blogs_train_large
rm ~/Data/blogs_train_large/6*
rm ~/Data/blogs_train_large/7*
```

接着，上传数据到亚马逊S3存储段。注意上传数据需要时间较长（好几百兆）。网速慢的读者，找个网快的地方上传比较好。

```
 aws s3 cp  ~/Data/blogs_train_large/ s3://ch12/blogs_train_large
recursive -exclude "*" --include "*.xml"
```

接下来使用mrjob连接亚马逊EMR——它帮助我们处理所有任务；只需要给出安全证书即可。按照下面链接给出的说明，为mrjob设置安全证书：https://pythonhosted.org/mrjob/guides/emr-quickstart.html。

设置完成后，修改mrjob的运行方式，让它在亚马逊EMR上运行，改动之处很少。用`-r`开关，告诉mrjob使用`emr`。然后把输入、输出目录改为我们在S3上创建好的存储段目录。虽然使用亚马逊云平台，但是运行时间仍很长。

```
python extract_posts.py -r emr s3://ch12gender/blogs_train_large/
outputdir=s3://ch12/blogposts_train/ --no-output
python nb_train.py -r emr s3://ch12/blogposts_train/ --output-dir=s3://
ch12/model/ --o-output
```

这次使用当然也是要付费的，但只有几美元[10]而已。如果你要持续运行或做其他跟大数据相关的任务，还请留意下费用问题。有一次我运行大量任务，花了20美元。我们这里任务量要小一些，估计花不到4美元。你可以查看下账户余额，设置收费提醒：https://console.aws.amazon.com/billing/home。

> 10澳元和美元货币符号都为$。作者表示写这本书时，两者汇率差别不大，但目前拉大了，实际费用用美元来计算比较合适。——译者注

没必要创建`blogposts_train`和存储模型文件的文件夹——EMR会自动创建。如果这两个文件夹存在，程序反而会报错。如果你再次运行代码的话，记得把这两个文件夹改成别的名字，但是要记得同时修改两条命令中的文件名（第一条命令中的输出目录是第二条命令的输入目录）。

如果等待时间过长，你有些厌烦，第一项任务运行一段时间后就可以停止。我建议至少运行15分钟后再停止，可能的话，最好1小时以上。你不能中断

第二项任务，中断后结果好不到哪去。第二项任务的运行时间大约是第一项任务的两到三倍。

回到S3控制界面，从存储段中下载输出的模型文件，将其保存到本地，我们就可以在笔记本文件中使用新模型。重新输入以下代码——与之前的不同之处用粗体表示，其实只更新了模型文件路径。

```
aws_model_filename = os.path.join(os.path.expanduser("~"), "models",
"aws_model")
aws_model = load_model(aws_model_filename)
y_true = []
y_pred = []
for actual_gender, predicted_gender in nb_predict_many(aws_model,
testing_filenames[0]):
    y_true.append(actual_gender == "female")
    y_pred.append(predicted_gender == "female")
y_true = np.array(y_true, dtype='int')
y_pred = np.array(y_pred, dtype='int')
print("f1={:.4f}".format(f1_score(y_true, y_pred, pos_label=None)))
```

使用更多数据后，我们发现F1值为0.81，结果较之前有所改善。

实验顺利完成后，如果不想继续支付费用，记得从亚马逊S3删除存储段——存储也是要钱的。

12.5　小结

本章研究的是如何运行大数据处理任务。其实，从各方面标准来看，我们的数据集还是有点小——只有几百兆。很多行业实际数据集都比这大多了，对其进行计算就需要更强大的计算能力。此外，我们所使用的算法可根据具体的任务做进一步优化，以提升扩展能力。

为了预测博主的性别，我们从博客文章中抽取词频特征。我们借助mrjob包，用映射和规约方法抽取博客内容和词频。抽取完成后，训练朴素贝叶斯分类器，预测博主性别。

我们可以用mrjob包先在本地测试，然后使用亚马逊的EMR云平台。你也可以使用其他云平台，甚至定制亚马逊EMR集群运行MapReduce任务，但是需要更多的操作才能跑起来。

附录　接下来的方向

全书正文部分至此已结束，学习本书的过程中，也许你已经注意到有很多路我们没有走，遇到多种选择的情况，我们可能只讲了其中几种，还有些主题没有充分展开。在附录部分，我整理了各章进一步学习方向，便于那些意犹未尽的读者深入学习，继续提升用Python挖掘数据的能力。如果你乐于接受挑战的话，本章不容错过。

附录按照章节来组织，每章给出与数据挖掘相关的文章、书籍或其他资源，同时还就该章挖掘任务提出新的挑战，有时只让读者做些小改进，但也给出了几个工作量比较大的挑战——对于难度较大的任务，介绍内容也会更多一些。

第 1 章——开始数据挖掘之旅

Scikit-learn教程

http://scikit-learn.org/stable/tutorial/index.html

scikit-learn文档提供了一系列数据挖掘教程。教程涵盖范围广，从适合新手把玩的数据集一直介绍到最近研究所用的技术。

想从头到尾学一遍这个教程，不肯下功夫是不行的——这份教程很全面——也值得在上面下功夫。

扩展IPython Notebook

http://ipython.org/ipython-doc/1/interactive/public_server.html

IPython Notebook功能强大，扩展方式很多，比如可以在别的计算机上创建服务器，运行笔记本文件，在你的主计算机上访问即可。如果你的主计算机性能一般，比如配置还算凑合的笔记本电脑，而其他可以供你支配的计算机性能很好，使用IPython Notebook在性能不错的计算机运行服务器是个好主意。此外，你还可以创建节点，采用并行方式处理数据。下面网站有不少数据集，一并分享给你。

http://archive.ics.uci.edu/ml/。

网上有很多由学术、商业和政府机构提供的数据集。UCI 机器学习数据库里有一系列标注好的数据集，用它们来测试算法很不错。

尝试下看看OneR算法在不同数据集上的表现。

第 2 章——用`scikit-learn`估计器分类

k近邻算法的扩展

https://github.com/jnothman/scikit-learn/tree/pr2532

k-近邻算法的简单实现速度很慢——它查看任意两个数据点是否足够近。当然还有更好的方法，`scikit-learn`实现了几种.。例如，我们可以用kd-tree提升算法速度。

另外一种提升搜索速度的方法叫作局部敏感散列（Locality-Sensitive Hashing，LSH）。这是对`scikit-learn`提出的一项改进，写作本书时，还没有加到`scikit`库中。上述链接为`scikit-learn`库的一个开发分支，你可以在数据集上测试LSH。具体方法还请参考该分支的相关文档。

复制代码仓库，按照下述网址给出的方法，安装最新版本：http://scikit-learn.org/stable/install.html。

记住要使用这个Git仓库的代码，而不要使用官方的代码。我建议你使用`virtualenv`或虚拟机安装，不要直接将其装到你的计算机上。这里有一份很棒的`virtualenv`教程：http://docs.python-guide.org/en/latest/dev/virtualenvs/。

更多复杂的流水线

http://scikit-learn.org/stable/modules/pipeline.html#featureunion-composite-feature-spaces

书中所用到的流水线都是线性的——上一步的输出作为下一步的输入。

流水线也遵循转换器和估计器的接口——流水线可以嵌套，方便构造更复杂的模型，再结合使用上述链接提到的FeatureUnion1方法，流水线将变得更加强大。

> 1FeatureUnion：将多个转换器对象结合在一起，组成一个新的转换器，汇总每个转换器的输出结果。——译者注

这样一次可以抽取多种类型的特征，组成新的数据集。更多细节及示例请见：http://scikit-learn.org/stable/auto_examples/feature_stacker.html。

比较分类器

scikit-learn提供了很多分类器。在选择分类器时，我们需要考虑一系列因素。你可以通过比较它们的F1值确定哪种效果更好。通过计算各F1值的标准差，就能确定几个分类器的分类效果之间的差异是否显著。

另外需要注意的是，几个分类器应该使用相同的训练集和测试集——也就是说，第一个分类器的测试数据，应该跟其他所有分类器的测试数据相同。使用相同的随机状态可以保证这一 点——对于重现实验结果很重要。

第 3 章——用决策树预测获胜球队

pandas的更多内容

http://pandas.pydata.org/pandas-docs/stable/tutorials.html

pandas库很周到——你加载数据时所能用到的各种功能，它很可能都已实现了。更多内容请见上面的链接。

Chris Moffitt 写了篇很不错的博文，介绍常见的Excel数据处理任务，如何用pandas完成：http://pbpython.com/excel-pandas-comp.html。

你还可以用pandas处理大数据集；可以看下StackOverflow用户Jeff对以下问题的解答（写作本书时2是最佳答案），了解处理过程：http://stackoverflow.com/questions/14262433/large-data-work-flows-using-pandas。

> 2翻译本书时依旧是最佳答案。——译者注

Brian Connelly的pandas教程也很棒：

http://bconnelly.net/2013/10/summarizing-data-in-python-with-pandas/。

更多复杂特征

http://www.basketball-reference.com/teams/ORL/2014_roster_status.html

每个赛季不同的比赛，上场队员也会有所不同。本来志在必得的比赛，很可能因为核心球员受伤而打得很艰难。你可以从篮球参考网站找到球队的队员名单。例如，上述链接指向的就是2013—2014赛季奥兰多魔术队（Orlando Magic）球员名单——所有NBA球队的类似数据都能从该网站找到。

编写代码，整合队员异动信息，以此作为新特征，可以有效提升模型表现，但是工作量不小！

第 4 章——用亲和性分析方法推荐电影

新数据集

http://www2.informatik.uni-freiburg.de/~cziegler/BX/

很多用于特定领域推荐任务的数据集，都值得探索。例如，Book-Crossing数据集包含来自27.8万用户的100多万条图书打分信息。有些评价信息很明确（用户给出分数），而有些则很模糊。模糊评价的权重不应该跟明确评价的一样高。

音乐网站www.last.fm公开了可用于音乐推荐的数据集：http://www.dtic.upf.edu/~ocelma/MusicRecommendationDataset/。

这里有一个用于笑话推荐的数据集！请见http://eigentaste.berkeley.edu/dataset/。

Eclat算法

http://www.borgelt.net/eclat.html

本章实现的Apriori算法是最常用的关联规则挖掘算法，但不是最好的。比较而言，Eclat算法更年轻也更强大，实现起来也比较容易。

第 5 章——用转换器抽取特征

增加噪音

本章讲到了如何通过删除噪音来提升性能；然而，有些数据集需要增加噪音。原因很简单——防止分类器过拟合训练数据，从而生成适用性更强的规则（但过多的噪音会导致模型过于宽泛，效果反而不好）。尝试实现往数据集中添加噪音的转换器。在UCI ML的数据集上进行测试，看看能否提升在测试集上的表现。

Vowpal Wabbit

http://hunch.net/~vw/

Vowpal Wabbit项目用于快速从文本中抽取特征。它针对Python进行了封装，可以直接在Python代码中调用。在大数据集上试试它的效果，比如第12章使用的博客语料。

第 6 章——使用朴素贝叶斯进行社会媒体挖掘[3]

[3]社会媒体挖掘相关技术可参考人民邮电出版社出版的《社会媒体挖掘》一书。——编者注

垃圾信息监测

http://scikit-learn.org/stable/modules/model_evaluation.html#scoring-parameter

我们用本章所学到的内容，就可以创建用来监测社会媒体广播是否为垃圾消息的应用。你可以尝试创建由垃圾广播/正常广播组成的数据集，实现文本挖掘算法，评估结果。

垃圾监测算法需要考虑假阳性和假阴性的情况。很多人宁愿多浏览几条垃圾信息，也不愿因为垃圾过滤器过于强大而错过合法的信息。为了满足人们这一需求，你可以把F1值作为评估标准，用网格搜索寻找合适的参数值，具体方法请见上面链接。

自然语言处理和词性标注

http://www.nltk.org/book/ch05.html

本章我们所用到的技术比起其他领域使用的复杂语言模型算是轻量级。例如，别的模型可能使用词性标注来消除同形异义词的歧义，提升准确性。NLTK配套的书4有一章专门讲这个问题，详细内容请见上面的链接。全书也很值得一读。

> 4中文版书名为《Python自然语言处理》，国内还可以找到英文版的影印版。据NLTK官网介绍2016年上半年将出版第2版。——译者注

第 7 章——用图挖掘找到感兴趣的人

更复杂的算法

https://www.cs.cornell.edu/home/kleinber/link-pred.pdf

关于如何预测包括社交网络在内的图结构中节点之间是否存在边，已经有大量相关研究。例如，David Liben-Nowell和Jon Kleinberg曾就这一主题发表了一篇论文，对于更复杂算法的设计很有帮助，论文请见上面的链接。

NetworkX

https://networkx.github.io/

如果你以后经常处理图和网络结构，深入研究NetworkX很有必要——可视化选项很好用，算法实现也非常不错。另外一个叫作SNAP的库也针对Python做了封装，请见http://snap.stanford.edu/snappy/index.html。

第 8 章——用神经网络破解验证码

好（坏？）验证码

http://scikit-image.org/docs/dev/auto_examples/applications/plot_geometric.html

我们所破解的验证码比起如今网站常用的要简单不少。你可以使用以下技巧来提升破解难度：

- 采用不同的转换方法。比如`scikit-image`所提供的（请见上

面链接）

- 使用不同的颜色；尝试无法很好转换为灰度模式的颜色
- 在图像中添加线条或其他形状：http://scikit-image.org/docs/dev/api/skimage.draw.html

深度网络

使用上述技巧产生的验证码会愚弄我们的破解工具，我们只好改进它。尝试使用第11章的深度网络。

网络越大，需要更多数据，因此你可能需要生成几千张验证码图像以达到好的效果。数据集的生成适合用并行方式来处理——因为它包含大量可以被分别处理的小任务。

增强学习

http://pybrain.org/docs/tutorial/reinforcement-learning.html

数据挖掘领域的下一个方向——增强学习势头日猛，虽然它的出现已经有一段时间了！PyBrain提供了几种增强学习算法，值得在刚创建的验证码数据集上一试（其他数据集也行！）。

第 9 章——作者归属问题

增加数据量

本章预测发件人的应用只使用了一部分邮件，数据集中还有很多邮件可以用。增加发件人的数量可能会导致正确率下降，但是使用相似的方法也有可能提升算法的表现。N元语法中的N以及参数到底取什么值，支持向量机能达到最佳效果，使用网格搜索就能解决这两个问题。我们争取在发件人数量很多的情况下，也能取得好的效果。

博客语料

第12章博客语料给出了博主编号（每个编号代表一个博主），把博主编号看成是类别，就能将其用作本章实验语料。此外，性别、年龄、行业和星座都可以作为分类任务来进行预测——解决作者归属的方法擅长处理这些分类问题吗？

局部N元语法

https://github.com/robertlayton/authorship_tutorials/blob/master/LNGTutorial.ipynb

另外一种分类器使用局部N元语法，从每个作者的作品而不是全部作者的作品中选择最佳特征。我写了一篇用局部N元语法解决作者归属问题的教程，请见上面链接。

第 10 章——新闻语料分类

算法评价

聚类算法效果评价是个大难题——一方面，我们也能多少知道好的分簇结果应该是什么样子；另一方面，如果我们确实知道，那就应该标注一部分数据，使用有监督的分类器对数据进行分类！该话题相关文章很多。下面这份文档介绍了聚类算法评价的难点所在：

http://www.cs.kent.edu/~jin/DM08/ClusterValidation.pdf

这里有份关于该主题的更全面（虽然有点过时）的一份文档：http://web.itu.edu.tr/sgunduz/courses/verimaden/paper/validity_survey.pdf。

`scikit-learn`实现了几种评价方法，对它们的综述请见http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation。

有了这些方法，你就能评估使用哪些参数值可以取得更好的聚类效果。网格搜索能帮助我们找到参数的最佳取值——就像在分类中用到的那样！

近期趋势分析

你可以运行我们本章编写的代码几个月。为期间发现的每个簇添加几个标签，便于我们了解哪些主题一直很活跃，从而了解近几个月全世界新闻焦点。

我们可以使用调整互信息等评价标准，比较聚类结果，请见上面评价方法综述链接。观察下一个月、两个月、半年和一年之后聚类的变化

结果。

实时聚类

k-means算法可以随着时间的推进，不断训练和更新，而不是在给定时间内做离散分析。你可以通过几种方法跟踪聚类变化——例如，跟踪每个簇中频率最高的几个单词或质心点每天移动的距离。别忘了网站API使用上限——你可能只需要每隔几个小时检查下有无新数据，就能保证结果是实时的。

第 11 章——用深度学习方法为图像中的物体进行分类

Keras和Pylearn2

如果你想进一步了解如何用Python做深度学习，Keras和Pylearn2这两个库有必要了解一下。它们都是以`Theano`为基础，但各有各的使用方法和特点。

Keras请见https://github.com/fchollet/keras/。

Pylearn2请见http://deeplearning.net/software/pylearn2/。

写作本书时，这两个库还不成熟，比较起来Pylearn2更稳定。它们都能很好完成你交给它们的任务，在做其他深度学习项目时，可以尝试用下。

另外一个很火的库叫Torch，但是写作本书时，还没有Python接口可用（请见http://torch.ch/）。

Mahotas

Mahotas图像处理包提供更好和更复杂的图像处理方法，使用它们能提升正确率，虽然计算量可能大幅上升。然而，很多图像处理任务适合并行处理，所以计算量加大也不再是问题。图像分类相关研究资料很多，下面这两份文档给出不少资料，可以先从这里找起：

http://luispedro.org/software/mahotas/.

http://ijarcce.com/upload/january/22-A%20Survey%20on

%20Image%20Classification.pdf

其他图像数据集有：

http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html

一些学术机构和行业组织也开放了很多图像数据集。上述网站介绍了很多数据集及使用它们的最好算法。要实现这样的一些好算法需要大量代码，但是回报远大于付出。

第 12 章——大数据处理

Hadoop课程

雅虎和谷歌面向不同水平的开发者提供了一系列非常实用的Hadoop课程。虽然侧重点不在Python，但是了解Hadoop概念后，再把它们应用到Pydoop或其他类似的库都能取得很好的效果。

雅虎的课程：https://developer.yahoo.com/hadoop/tutorial/。

谷歌的课程：https://cloud.google.com/hadoop/what-is-hadoop。

Pydoop

Pydoop是一个用来运行Hadoop任务的Python库——教程请见http://crs4.github.io/pydoop/tutorial/index.html。

Pydoop可以跟HDFS配合使用，mrjob有类似功能，但在运行某些任务时，Pydoop能给你更多的控制权。

推荐引擎

创建大型的推荐引擎是测试你大数据技能的好方法。Mark Litwintschik在他的博文中介绍了如何使用大数据技术Apache Spark创建推荐引擎，详见http://tech.marksblogg.com/recommendation-engine-spark-python.html。

更多资源

Kaggle竞赛：www.kaggle.com/

Kaggle定期举行数据挖掘竞赛，一般都会提供奖金。参加Kaggle竞赛是学习处理真实场景数据挖掘任务的好方法。他们的社区用户友善、乐于分享——通常你能见到竞赛前10名选手分享代码！

Coursera在线学习网站：

www.coursera.org

Coursera网站上有很多关于数据挖掘和数据科学的课程，大多是专注于某个方向，比如有专门讲大数据和图像处理的。吴恩达（Andrew Ng）老师的机器学习课程综合性强，涉及面比较广，这门课程很受欢迎，从它开始学起很不错：https://www.coursera.org/learn/machine-learning/。

这门课所讲的内容比本书难度要大一点，感兴趣的读者可以将其作为今后的学习方向。

神经网络方面，请留意下这门课程：https://www.coursera.org/course/neuralnets。

如果你完成了上述课程的学习，可以关注下这门关于概率图模型的课程：https://www.coursera.org/course/pgm。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_interview，讲述码农精彩人生
- 微信 图灵教育：turingbooks

091507240605ToBeReplacedWithUserId

版权信息

书名：Hadoop安全：大数据平台隐私保护

作者：[美] Ben Spivey Joey Echeverria

译者：赵双　白波

ISBN：978-7-115-46771-3

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

091507240605ToBeReplacedWithUserId

版权声明
O'Reilly Media, Inc. 介绍
业界评论
序
前言
目标读者
排版约定
使用代码示例
Safari® Books Online
联系我们
致谢
来自Joey的致谢
来自Ben的致谢
来自Eddie的致谢
免责声明
电子书
第 1 章　引言
1.1　安全概览
1.1.1　机密性
1.1.2　完整性
1.1.3　可用性
1.1.4　验证、授权和审计
1.2　Hadoop安全：简史
1.3　Hadoop组件和生态系统
1.3.1　Apache HDFS
1.3.2　Apache YARN
1.3.3　Apache MapReduce
1.3.4　Apache Hive
1.3.5　Cloudera Impala
1.3.6　Apache Sentry
1.3.7　Apache HBase
1.3.8　Apache Accumulo

1.3.9 Apache Solr
1.3.10 Apache Oozie
1.3.11 Apache ZooKeeper
1.3.12 Apache Flume
1.3.13 Apache Sqoop
1.3.14 Cloudera Hue
1.4 小结
第一部分 安全架构
第 2 章 保护分布式系统
2.1 威胁种类
2.1.1 非授权访问 / 伪装
2.1.2 内在威胁
2.1.3 拒绝服务
2.1.4 数据威胁
2.2 威胁和风险评估
2.2.1 用户评估
2.2.2 环境评估
2.3 漏洞
2.4 深度防御
2.5 小结
第 3 章 系统架构
3.1 运行环境
3.2 网络安全
3.2.1 网络划分
3.2.2 网络防火墙
3.2.3 入侵检测和防御
3.3 Hadoop角色和隔离策略
3.3.1 主节点
3.3.2 工作节点
3.3.3 管理节点
3.3.4 边界节点
3.4 操作系统安全

3.4.1　远程访问控制
3.4.2　主机防火墙
3.4.3　SELinux
3.5　小结
第 4 章　Kerberos
4.1　为什么是Kerberos
4.2　Kerberos概览
4.3　Kerberos工作流：一个简单示例
4.4　Kerberos信任
4.5　MIT Kerberos
4.5.1　服务端配置
4.5.2　客户端配置
4.6　小结
第二部分　验证、授权和审计
第 5 章　身份和验证
5.1　身份
5.1.1　将Kerberos主体映射为用户名
5.1.2　Hadoop用户到组的映射
5.1.3　Hadoop用户配置
5.2　身份验证
5.2.1　Kerberos
5.2.2　用户名和密码验证
5.2.3　令牌
5.2.4　用户模拟
5.2.5　配置
5.3　小结
第 6 章　授权
6.1　HDFS授权
HDFS扩展ACL
6.2　服务级授权
6.3　MapReduce和YARN的授权
6.3.1　MapReduce（MR1）

6.3.2　YARN (MR2)
6.4　ZooKeeper ACLs
6.5　Oozie授权
6.6　HBase和Accumulo的授权
6.6.1　系统、命名空间和表级授权
6.6.2　列级别和单元级别授权
6.7　小结
第 7 章　Apache Sentry（孵化中）
7.1　Sentry概念
7.2　Sentry服务
Sentry服务配置
7.3　Hive授权
Hive Sentry的配置
7.4　Impala授权
Impala的Sentry配置
7.5　Solr授权
Solr的Sentry配置
7.6　Sentry特权模型
7.6.1　SQL特权模型
7.6.2　Solr特权模型
7.7　Sentry策略管理
7.7.1　SQL命令
7.7.2　SQL策略文件
7.7.3　Solr策略文件
7.7.4　策略文件的验证和校验
7.7.5　从策略文件迁移
7.8　小结
第 8 章　审计
8.1　HDFS审计日志
8.2　MapReduce审计日志
8.3　YARN审计日志
8.4　Hive审计日志

8.5 Cloudera Impala审计日志
8.6 HBase审计日志
8.7 Accumulo审计日志
8.8 Sentry审计日志
8.9 日志聚合
8.10 小结
第三部分 数据安全
第 9 章 数据保护
9.1 加密算法
9.2 静态数据加密
9.2.1 加密和密钥管理
9.2.2 HDFS静态数据加密
9.2.3 MapReduce2中间数据加密
9.2.4 Impala磁盘溢出加密
9.2.5 全盘加密
9.2.6 文件系统加密
9.2.7 Hadoop中重要数据的安全考虑
9.3 动态数据加密
9.3.1 传输层安全
9.3.2 Hadoop动态数据加密
9.4 数据销毁和删除
9.5 小结
第 10 章 数据导入安全
10.1 导入数据的完整性
10.2 数据导入的机密性
10.2.1 Flume加密
10.2.2 Sqoop加密
10.3 导入工作流
10.4 企业架构
10.5 小结
第 11 章 数据提取和客户端访问安全
11.1 Hadoop命令行接口

11.2 保护应用安全
11.3 HBase
11.3.1 HBase shell
11.3.2 HBase REST网关
11.3.3 HBase Thrift网关
11.4 Accumulo
11.4.1 Accumulo shell
11.4.2 Accumulo代理服务
11.5 Oozie
11.6 Sqoop
11.7 SQL访问
11.7.1 Impala
11.7.2 Hive
11.8 WebHDFS/HttpFS
11.9 小结
第 12 章 Cloudera Hue
12.1 Hue HTTPS
12.2 Hue身份验证
12.2.1 SPNEGO后端
12.2.2 SAML后端
12.2.3 LDAP后端
12.3 Hue授权
12.4 Hue SSL客户端配置
12.5 小结
第四部分 综合应用
第 13 章 案例分析
13.1 案例分析：Hadoop 数据仓库
13.1.1 环境搭建
13.1.2 用户体验
13.1.3 小结
13.2 案例分析：交互式HBase Web应用
13.2.1 设计与架构

13.2.2 安全需求
13.2.3 集群配置
13.2.4 实现中的注意事项
13.2.5 小结
后记
统一授权
数据管控
原生数据保护
结语
关于作者
关于封面

版权声明

Copyright © 2015 Joseph Echeverria and Benjamin Spivey.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc.and Posts & Telecom Press, 2017. Authorized translation of the English edition, 2017 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版 2015 。

简体中文版由人民邮电出版社出版，2017。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media,Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。 O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 **O'Reilly** 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，

并且切实地按照 **Yogi Berra** 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，**Tim** 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

序

近来，“Hadoop 安全”成为了一个充满矛盾的名词。雅虎和 Facebook 等公司创建和使用的这个大数据平台的早期版本，在保护其存储的数据方面并没有花费太多精力。实际上也没有必要这么做，因为 Hadoop 里几乎没有什么敏感数据，只有对黑客毫无吸引力的状态更新和新闻报道。而且，找到它们也不需要花费很多精力。

然而，当这种平台被更多传统企业使用时，便开始涉及更多传统企业数据。金融交易、个人银行账号和税务信息、医疗记录等诸如此类的信息，正是黑客们所追逐的。Hadoop 如今广泛用于零售、银行和医疗保健行业，它也逐渐引来了小偷们的注意。

如果说数据是有吸引力的目标，那么大数据的规模和吸引力都是最大的。与之前的任何预处理系统相比，Hadoop 能从更多地方收集更多数据，然后将它们结合起来，用更多方法进行分析。Hadoop 正是通过这种方式创造了巨大的价值。

显然，“Hadoop 安全”意义重大。

本书的两位作者在将安全概念引入 Hadoop 平台方面做出过突出贡献。他们在书中阐述了 Hadoop 从早期开放的消费互联网时代到现在作为敏感数据可信平台的演变历程，回顾了 Hadoop 安全的历史，既涵盖了其优点和演进过程，也涉及了新的业务问题。身份验证、加密、密钥管理和商业实践在内的诸多主题及其在实际环境下的应用也在书中有详细讨论。

10 年前，Hadoop 被 Facebook 选作图片存储软件，它发展到今天的过程非常有趣。如今，Hadoop 为数据处理和分析提供了更强的能力、更多的方法、更大的规模和更好的表现。因此，无论分开来看还是整体观之，Hadoop 也需要更多保护。

本书最好的地方在于，不仅仅对问题进行描述，还给出了处理建议。它能清晰并且详细地告诉读者，搭建和使用 Hadoop 的资深开发者们如何安全地管理大数据。本书给读者提供了如何使用——并且安全地使用——这个先进的平台以分析、处理和理解数据的最佳建议。

——Mike Olson

Cloudera 公司首席战略官、联合创始人

前言

虽然 Apache Hadoop 仍然是一个相对较新的技术，但这并没有限制它迅速地被业界采用，并爆发式地出现了很多相关工具，正是这些工具组成了 Hadoop 广阔的生态系统。对于 Hadoop 用户，这当然是个令人振奋的时代。Hadoop 给公司带来了前所未有的增值机会，同时也给负责数据访问安全和系统合规的技术人员带来了很多挑战。目前，已经有丰富的信息能够帮助使用 Hadoop 构建解决方案的开发者，以及部署运维 Hadoop 的管理员。然而，关于如何设计和实现 Hadoop 安全部署的指导信息依然匮乏。

本书全面而深入地介绍了 Hadoop 的众多安全特性，并使用通用的计算机安全概念进行组织。第 1 章是介绍性内容，随后分为 4 大部分：第一部分是安全架构，第二部分是验证、授权和安全审计，第三部分是数据安全，第四部分是归纳总结。这些部分涵盖了从设计安全架构的前期步骤，到实现通用的安全访问控制和数据保护。最后介绍了几个用例，融合了书中的很多概念。

目标读者

本书的目标人群是管理大数据平台安全的 Hadoop 管理员，以及需要在大型企业架构中设计集成 Hadoop 安全规划的安全架构师。书中介绍了很多 Hadoop 安全概念，包括验证、授权、审计、加密和系统架构。

第 1 章是对贯穿全书的一些安全概念的综述，以及对 Hadoop 生态系统的简介。如果刚刚接触 Hadoop，建议你阅读《Hadoop 技术详解》和《Hadoop 权威指南》。本书假设读者熟悉 Linux、计算机网络以及一般的系统架构知识。对于那些在保护分布式系统安全方面没有经验的管理员，第 2 章会给出这部分内容的概述。有经验的安全架构师如果不想回顾这方面知识，可以跳过这一章。总体而言，本书不要求读者具备编程背景，而尽量将内容重点集中于 Hadoop 安全在架构和操作方面的实现。

排版约定

本书使用以下排版约定。

- 楷体

  表示新术语和重点强调的内容。

- 等宽字体（`constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键词等。

- 加粗等宽字体（**`constant width bold`**）

  表示命令以及其他需要用户输入的文字。

- 等宽斜体（*`constant width italic`*）

  表示这些值应该替换为用户输入，或根据上下文确定。

 该图标表示提示或建议。

 该图标表示一般性说明。

 该图标表示警告或警示。

使用代码示例

我们在全书提供了配置文件，用以指导读者为自己的 Hadoop 环境进行安全防护。这些样例中，可供下载的版本链接为：https://github.com/hadoop-security/examples。在第 13 章，本书提供了设计、实现和部署一个保存网页快照的网络接口的完整样例。该样例的完整源代码以及关于可部署应用的 Hadoop 集群的安全配置说明，都可以在 GitHub 上下载（https://github.com/hadoop-security/kite-spring-hbase-example）。

本书旨在帮助读者完成自己的工作。通常情况下，如果书中包含代码样例，你可以在自己的程序和文档中使用它们，不需要联系我们申请授权，除非需要直接复制相当一部分的代码。例如，编写程序时，使用本书中的几个代码片段并不需要申请授权，但出售或分发 O'Reilly 图书代码样例的光盘则需要获得许可；引用本书或引用样例解答问题

不需要授权，但将本书的大量样例代码纳入产品的文档则需要获得许可。

我们不强制要求你在引用本书内容时进行声明，但如果你这么做，我们会非常感激。引用信息通常包括书名、作者、出版社和 ISBN，如：*Hadoop Security* by Ben Spivey and Joey Echeverria (O'Reilly). Copyright 2015 Ben Spivey and Joey Echeverria, 978-1-491-90098-7。

若你认为对样例代码的使用需要授权，请通过这个邮箱联系我们：permissions@oreilly.com。

Safari® Books Online



Safari Books Online 是应运而生的数字图书馆，它同时以图书和视频的形式出版世界顶级技术和商业作家的专业作品。

技术专家、软件开发人员、Web 设计师、商务人士和创新专家等，都将 Safari Books Online 作为开展调研、解决问题、学习和认证培训的首选资源。

Safari Books Online 为企业、政府、教育和个人提供各种产品组合和灵活的定价策略。

会员可以通过搜索，从数据库中访问数以千计的图书、培训视频和正式出版前的书稿，这些数据来源包括 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、 Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。要了解 Safari Books Online 的更多信息，请访问我们的网站（http://www.safaribooksonline.com/）。

联系我们

请把对本书的评价和问题发给出版社。美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

我们为本书提供了专门网页，上面有勘误表、示例以及其他信息。可以通过 http://bit.ly/hadoop-security 访问该网页。本书中文版勘误可到 http://www.ituring.com.cn/book/1600 提交。

为本书提供建议或咨询技术问题，请发邮件到 bookquestions@oreilly.com。

想了解更多关于 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com。

我们的其他联系方式如下：

Facebook：http://facebook.com/oreilly

Twitter：http://twitter.com/oreillymedia

YouTube：http://www.youtube.com/oreillymedia

致谢

Ben 和 Joey 要感谢以下所有人：编辑 Marie Beaugureau 和所有 O'Reilly 公司的工作人员；Ann Spencer；贡献了友情客串章节的 Eddie Garcia；主要技术审阅者 Patrick Angeles、 Brian Burton、Sean Busbey、Mubashir Kazia 和 Alex Moundalexis；Jarek Jarcec Cecho；提供宝贵意见的其他作者 Eric Sammer、Lars George 和

Tom White；还有为我们提供集体支持的 Cloudera 的伙伴们，本书在他们的帮助下才得以问世。

来自Joey的致谢

我想将此书献给 Maria Antonia Fernandez、Jose Fernandez 和 Sarah Echeverria，他们在此前的每个日子里都鼓舞着我，让我觉得自己能完成想要完成的任何事情。我还要感谢我的父母 Maria 和 Fred Echeverria，我的兄弟姐妹 Fred、Marietta、Angeline、Paul Echeverria 和 Victoria Schandevel，他们始终给我爱与支持。此外，若是没有 Apache Hadoop 社区难以置信的强大支持，我无法完成本书。这里我无法列出帮助我们的每个人，不过 Ben 在后面列出了一些——当然不是全部。最后，我想感谢我的合著者 Ben：我们共同完成了一件了不起的事情，Bennie（Paul，别客气）。

来自Ben的致谢

我想以此书纪念 Ginny Venable 和 Rob Trosinski，我深深地怀念他们。我想感谢妻子 Theresa，是你给予我无限支持和理解，以及始终让我微笑的 Oliver Morton。感谢我的父母 Rich 和 Linda，谢谢你们长期以来向我展示教育的价值，并以卓越的专业才能树立了榜样。感谢 Matt、Jess、Noah，以及 Spivey 家庭的其他成员，感谢 Mary、Jarrod 和 Dolly Trosinsk，感谢 Swope 一家，以及整个过程中给予我极大帮助的朋友们：Hemal Kanani (BOOM)、Ted Malaska、Eric Driscoll、Paul Beduhn、Kari Neidigh、Jeremy Beard、Jeff Shmain、Marlo Carrillo、Joe Prosser、Jeff Holoman、Kevin O'Dell、Jean-Marc Spaggiari、 Madhu Ganta、Linden Hillenbrand、Adam Smieszny、Benjamin Vera-Tudela、Prashant Sharma、Sekou Mckissick、Melissa Hueman、Adam Taylor、Kaufman Ng、Steve Ross、Prateek Rungta、Steve Totman、Ryan Blue、Susan Greslik、Todd Grayson、Woody Christy、Vini Varadharajan、Prasad Mujumdar、Aaron Myers、Phil Langdale、Phil Zeyliger、Brock Noland、Michael Ridley、Ryan Geno、Brian Schrameck、Michael Katzenellenbogen、Don Brown、Barry Hurry、Skip Smith、Sarah Stanger、Jason Hogue、Joe Wilcox、Allen Hsiao、Jason Trost、Greg Bednarski、Ray Scott、Mike Wilson、Doug Gardner、Peter Guerra、Josh Sullivan、Christine Mallick、Rick Whitford、Kurt Lorenz、Jason Nowlin 和 Chuck Wigelsworth。最后，我要感谢 Joey，还好他帮我一起写了这本书

——我从不认为能够独自完成本书。如果我由于疏忽忘记了一些朋友，请接受我最诚挚的歉意。

来自**Eddie**的致谢

我想感谢我的家人和朋友，他们在我第一本书的写作历程中给予了支持和鼓励。Sandra、Kassy、Sammy、Ally、Ben、Joey、Mark 和 Peter，谢谢你们。

免责声明

感谢你阅读本书。虽然笔者试图对 Hadoop 生态系统的不同安全特性进行解释、文档化并给出建议，但这并不表明或暗示，使用这些特性中的任何一个将获得完全安全的集群。从安全角度来说，无论使用什么保护机制，任何信息系统都不是 100% 安全的。我们鼓励对 Hadoop 环境进行持续的安全检查，从而确保可能达到的最好的安全状态。对于可能由于使用本书中任何特性所造成的损坏，笔者、O'Reilly 公司及人民邮电出版社不对此负责。所有操作，风险自负。

电子书

扫描如下二维码，可购买本书电子版。



第 1 章　引言

早在 2003 年，谷歌就发表了一篇论文（http://research.google.com/archive/gfs.html），阐述了基于服务器集群的、面向海量数据存储的可扩展系统架构，该架构称为谷歌文件系统（**GFS**）。一年后，谷歌发表了另一篇论文（http://research.google.com/archive/mapreduce.html），阐述了一个名为 **MapReduce** 的编程模型。该模型利用 GFS 并行处理数据，实现了处理程序在数据存储处的本地化运行。几乎同时，Doug Cutting 与其他合作者也在搭建一个开源的网络爬虫，如今这个工具被称为 Apache Nutch（http://nutch.apache.org）。Nutch 的开发者们意识到，MapReduce 编程模型和 GFS 是很完美的分布式网络爬虫构建模块，于是他们着手实现基于这两个项目的新爬虫版本。这些组件后来从 Nutch 脱离，并组成了 Apache Hadoop 项目。围绕 Hadoop 可扩展架构形成的生态系统 1 允许用户存储和处理所有重要的事务数据，提供了解决问题的另一种方法。

> 1Apache Hadoop 本身由 4 个子项目组成：HDFS、YARN、MapReduce 和 Hadoop Common。然而， Hadoop 生态系统、Hadoop 和其他在 Hadoop 基础上创建或集成的相关项目通常均被简称为 Hadoop。我们此处使用 Hadoop 项目和 Hadoop 生态系统以示区分。

当所有这些崭新且令人激动的处理、存储数据的方法在 Hadoop 生态系统中被广泛使用时，使用单一的集中式集群管理这些 PB 级数据明显是有风险的。成百上千的服务器在同一个通用应用栈中互连，给如何保护这些重要资源提出了很多难题。其他书籍侧重于介绍如何编写 MapReduce 代码、如何设计最优的导入框架，以及如何在 Hadoop 生态系统上搭建复杂的低延迟处理系统，而本书专注于讨论在 Hadoop 的安全架构下，如何利用众多安全特性保护上述所有事务的安全性。

1.1　安全概览

开始讨论 Hadoop 专有内容之前，有必要先了解一些信息安全相关的关键理论和术语。信息安全理论的核心是 **CIA** 模型，即机密性（**confidentiality**）、完整性（**integrity**）和可用性（**availability**）。该模型的这 3 个组件是高层次概念，能够广泛应用于信息系统、计算平台以及（更适用于本书的）Hadoop。我们还

将进一步了解验证、授权和审计这 3 个安全计算领域的要素，并将在全书深入探讨。

> 虽然 CIA 模型能够帮助建立一些信息安全原则，但需要指出的是，该模型并不是一个需要严格遵循的规范。Hadoop 平台的安全特性既有可能涉及多个 CIA 模型组件，也有可能连一个 CIA 组件也覆盖不到。

1.1.1 机密性

机密性这个安全原则强调，信息只应该被期望的接收者看到。例如，假设 Alice 通过邮件向 Bob 发送了一封信，那么只有当 Bob 是唯一能够阅读这封信的人时，才能认为这封信是机密的。尽管这看起来很简单，不过要保证机密性，还需要几个重要的安全概念。比如，如何使 Alice 知道她发送的信件被 Bob 本人读了？如果 Bob 本人读了信件，他如何确认自己所读的信是 Alice 本人发的？为了使 Alice 和 Bob 都参与机密信息的传递，他们需要持有能够将自己和其他人区别开来的唯一身份标识。此外，Alice 和 Bob 需要通过一个身份验证的过程证明自己的身份。身份和验证是 Hadoop 安全的核心组件，这部分将在第 5 章详细介绍。

机密性的另一个重要概念是加密。加密是这样一种机制，它将数学算法应用于信息片段，使加密后的输出内容对于非预期接收者不可读。只有期望的接收者能对加密消息进行解密，从而得到原始信息。数据加密有两种方式：静态加密和动态加密。静态加密是指，数据在非访问状态下就处于加密状态。例如，存储在硬盘上的加密文件就是静态加密的一个例子。动态加密也称在线加密，即在数据通过网络从一个地方发送到另外一个地方时进行加密。两种加密方式可以各自独立使用，也可以混合使用。第 9 章将介绍 Hadoop 的静态加密，而动态加密会在第 10 章和第 11 章进行介绍。

1.1.2 完整性

完整性是信息安全的重要组成部分。之前的例子中，当 Alice 发送信件给 Bob 时，如果 Charles 在 Alice 和 Bob 毫不知情的情况下，于传输过程中截获了信件，并对其内容进行了修改，这会怎样呢？ Bob 如何保证他接收到的信件和 Alice 发送的是完全一样的呢？这就是数据完整性的概念。数据完整性是信息安全的关键要素，尤其在包含高度敏感数据的行业中。设想一下，如果银行没有机制能够验证客户账户

余额完整性，会发生什么？医院没有机制能够验证病人病历完整性，又会发生什么？政府没有验证情报秘密完整性的机制，可能会发生什么？即使机密性得到保证，但完整性得不到保证的数据也有被损坏的风险。完整性将在第 9 章和第 10 章讲解。

1.1.3 可用性

可用性与前两个要素不属于同一类规范。机密性和完整性同是众所周知的安全概念，而可用性更多涉及的是操作准备。例如，如果 Alice 试图向 Bob 发信，但邮局关门了，那么信件就无法被送达。这样对于 Bob 来说，信件就是不可用的。数据或服务的可用性能够被普通的行为影响，如计划内的停工升级或应用安全补丁，同时也可能被诸如分布式拒绝服务攻击（DDoS）等安全事件影响。《Hadoop 技术详解》和《Hadoop 权威指南》介绍了如何对高可用性进行配置，本书将从安全视角介绍这些概念，参见第 3 章和第 10 章。

1.1.4 验证、授权和审计

验证、授权和审计（通常缩写为 AAA）指计算机安全领域的一个架构模式。在该模式中，使用服务的用户先要证明自己的身份，然后根据规则被授予权限，同时其操作被记录下来留待审计。与 AAA 紧密相联的一个概念是身份。身份是指系统如何将不同的实体、用户和服务区分开来，典型的代表是一个任意字符串，例如用户名或者诸如用户 ID（UID）的唯一号码。

深入了解 Hadoop 如何支持身份、认证、授权和审计之前，先考虑在一个很简单的情况下运用这些概念：在单台 Linux 服务器上使用 sudo 命令。我们来看看两个不同用户 Alice 和 Bob 进行的终端会话。在这台服务器上，Alice 的用户名是 *alice*，Bob 的用户名是 *bob*。如示例 1-1 所示，Alice 先登录。

示例 **1-1** 验证和授权

```
$ ssh alice@hadoop01
alice@hadoop01's password:
Last login: Wed Feb 12 15:26:55 2014 from 172.18.12.166
[alice@hadoop01 ~]$ sudo service sshd status
openssh-daemon (pid 1260) is running...
[alice@hadoop01 ~]$
```

示例 1-1 中，Alice 通过 SSH 登录，并且立即被提示输入她的口令。

她的用户名 / 口令对被用于在 /etc/passwd 密码文件中验证其登录。完成这一步后，Alice 便被成功验证了用户名 alice 的身份。Alice 的下一步是使用 sudo 命令获得 sshd 服务，这需要超级用户的权限。这个命令执行成功，说明 Alice 被授权可以执行该命令。对于 sudo 命令，控制哪些用户能够被授予超级用户权限的规则存储在 /etc/sudoers 文件中，如示例 1-2 所示。

示例 **1-2** /etc/sudoers

```
[root@hadoop01 ~]# cat /etc/sudoers
root ALL = (ALL) ALL
%wheel ALL = (ALL) NOPASSWD:ALL
[root@hadoop01 ~]#
```

示例 1-2 中，用户 root 被授予使用 sudo 执行任何命令的权限，用户组 wheel 被授予可无密码使用 sudo 执行任何命令的权限。这个例子中，系统的验证依赖于登录，而非一次新的验证要求。最后一个问题是，系统怎么知道 Alice 是 wheel 用户组的一员呢？在 UNIX 和 Linux 系统中，这通常由文件 /etc/group 控制。

现在，我们知道控制 Alice 的身份标识的有两个文件：/etc/passwd（示例 1-4）和 /etc/group（示例 1-3）。/etc/passwd 文件为她的用户名分配一个特有的 UID 和主目录，/etc/group 文件更进一步定义了系统的组标识以及用户和组的从属关系。这些身份标识信息文件会在执行 sudo 命令时用到，同时用到的还有 /etc/sudoers 文件中定义的权限规则，它们用于验证 Alice 是否有权限执行请求的命令。

示例 **1-3** /etc/group

```
[root@hadoop01 ~]# grep wheel /etc/group
wheel:x:10:alice
[root@hadoop01 ~]#
```

示例 **1-4** /etc/passwd

```
[root@hadoop01 ~]# grepalice /etc/passwd
alice:x:1000:1000:Alice:/home/alice:/bin/bash
[root@hadoop01 ~]#
```

现在看看示例 1-5 中 Bob 的会话。

示例 **1-5**　授权失败

```
$ ssh bob@hadoop01
bob@hadoop01's password:
Last login: Wed Feb 12 15:30:54 2014 from 172.18.12.166
[bob@hadoop01 ~]$ sudo service sshd status

We trust you have received the usual lecture from the local System
Administrator. It usually boils down to these three things:

    #1) Respect the privacy of others.
    #2) Think before you type.
    #3) With great power comes great responsibility.

[sudo] password for bob:
bob is not in the sudoers file.  This incident will be reported.
[bob@hadoop01 ~]$
```

本示例中，Bob 能够使用与 Alice 类似的方法进行身份验证，但当他试图使用 `sudo` 命令时，遇见了截然不同的状况。首先，Bob 执行命令时被系统再次要求输入口令；成功提交口令后，被系统拒绝以超级用户的权限执行该 `service` 命令。这是由于与 Alice 不同，Bob 不是 `wheel` 组的成员，因而不被授权执行 `sudo` 命令。

了解了身份、验证和授权，我们再来关注审计。对于用户与诸如 SSH 和 sudo 等安全服务的互动行为，Linux 会创建一个名为 `/var/log/secure` 的日志文件。该文件能够记录一个账户的某些行为，无论这些行为成功还是失败。若我们在 Alice 和 Bob 各自完成上述操作后查看该日志，能够看到示例 1-6 所示的输出（按照方便阅读的形式组织）。

示例 **1-6**　/var/log/secure

```
[root@hadoop01 ~]# tail -n 6 /var/log/secure
Feb 12 20:32:04 ip-172-25-3-79 sshd[3774]: Accepted password for
  alice from 172.18.12.166 port 65012 ssh2
Feb 12 20:32:04 ip-172-25-3-79 sshd[3774]: pam_unix(sshd:session):
  session opened for user alice by (uid=0)
Feb 12 20:32:33 ip-172-25-3-79 sudo:    alice : TTY=pts/0 ;
  PWD=/home/alice ; USER=root ; COMMAND=/sbin/service sshd status
Feb 12 20:33:15 ip-172-25-3-79 sshd[3799]: Accepted password for
  bob from 172.18.12.166 port 65017 ssh2
Feb 12 20:33:15 ip-172-25-3-79 sshd[3799]: pam_unix(sshd:session):
  session opened for user bob by (uid=0)
```

```
Feb 12 20:33:39 ip-172-25-3-79 sudo:      bob : user NOT in sudoers;
  TTY=pts/2 ; PWD=/home/bob ; USER=root ; COMMAND=/sbin/service sshd statu
[root@hadoop01 ~]#
```

对于 Alice 和 Bob，他们通过 SSH 成功登录的事实均被记录下来，包括他们对 sudo 的使用记录。Alice 的例子中，系统记录了她成功使用 sudo 以 `root` 权限执行了 `/sbin/service sshd status` 命令。而 Bob 的例子中，系统则记录了他试图以 `root` 权限执行 `/sbin/service sshd status` 命令，但由于他不在 `/etc/sudoer` 用户组里，以致被系统拒绝执行。

该例子展示了身份、验证、授权和审计的概念如何运用在相对简单的单台 Linux 服务器上，并维持系统安全。这些概念将在第二部分的 Hadoop 相关章节中予以详细阐述。

1.2 Hadoop安全：简史

Hadoop 在存储和处理大量数据时效率很高，并且与其他平台相比更经济。Hadoop 项目最初的关注点是实际的技术实现，项目中许多代码涵盖的逻辑能够针对分布式系统中固有的复杂性进行处理，如故障处理与协同。由于这种较强的针对性，早期的 Hadoop 项目建立了一个安全立场：整个机器集群和所有访问它的用户都是可信网络的一部分。这实际上意味着，Hadoop 并没有强安全策略强制执行很多措施。

随着 Hadoop 项目的发展，显然，至少应当有一个机制能够对用户身份进行强有力的验证。Kerberos 被选作该项目的这种机制，它是一个完善的协议，如今广泛应用于 Microsoft Active Directory 等企业系统。在强认证策略之外，还要有强授权策略。强授权策略定义了单个用户被认证后能够做的事情。起初，授权策略是在单个组件上实现的，这意味着管理员需要在很多地方对授权控制进行定义。后来，在仍是孵化项目的 Apache Sentry 中，授权机制终于变得容易了一些。但就像我们将在第 6 章和第 7 章中看到的一样，目前尚未有一个能在整个生态系统中具有整体统筹性的授权机制。

Hadoop 安全的另一个仍在演变的方面是，通过加密和其他机密性机制进行的数据保护。在可信网络中，人们起初假定数据本来就是被保护的，不会被非授权用户访问，因为只有授权用户才能接入可信网络。之后，Hadoop 为节点间的数据传输添加了加密手段，对硬盘上的数据存储也进行了加密。在后续的讲解中，你能看到这些安全策略

的进化是如何产生的，但为了快速入门，首先还是需要关注 Hadoop 的生态系统。

1.3　Hadoop组件和生态系统

本节，我们将纵览贯穿全书的 Hadoop 生态系统的各个组件，这有助于在开始讨论组件的安全之前先了解它们。已经精通下列组件的读者可以直接跳至下一节。除非另外声明，本书描述的安全特性适用的项目版本均如表 1-1 所示。

表**1-1**：项目版本[a]

项目	版本
Apache HDFS	2.3.0
Apache MapReduce (for MR1)	1.2.1
Apache YARN (for MR2)	2.3.0
Apache Hive	0.12.0
Cloudera Impala	2.0.0
Apache HBase	0.98.0
Apache Accumulo	1.6.0
Apache Solr	4.4.0
Apache Oozie	4.0.0
Cloudera Hue	3.5.0
Apache ZooKeeper	3.4.5
Apache Flume	1.5.0
Apache Sqoop	1.4.4
Apache Sentry (Incubating)-1.4.0	incubating

a 细心的读者会发现，上述列表有一些遗漏。我们特意没有在上表中提到 Apache Spark、Apache Ranger 和 Apache Knox。这些项目是 Hadoop 生态系统新增的组件，由于时间的限制，我们在这里忽略了它们。

1.3.1　Apache HDFS

Hadoop 分布式文件系统（**HDFS**）通常被认为是 Hadoop 生态系统中其余部分的基础。HDFS 是 Hadoop 的存储层，在系统存储容量和带宽总量线性增长时，能够对海量数据进行存储。HDFS 是能够跨越多个服务器的逻辑文件系统，每个服务器都可有多块硬盘。从安全角度理解这一点很重要，因为 HDFS 中的某个文件可以跨越 Hadoop 集群中的多个或全部服务器。这意味着，客户端与某文件的交互有可能需要与集群中的每一个节点进行通信。HDFS 的一个关键实现方法是，将文件分解成块（**block**），这使得前述的交互成为可能。每一

个文件块被存储在集群中任何节点的任何物理驱动器上。此问题较为复杂，我们在这里不深入讨论，并忽略原理部分的细节。推荐有兴趣的读者阅读《Hadoop 权威指南》。关键的安全要点是：HDFS 中的所有文件都被分解为块，当使用 HDFS 的客户端读写这些文件时，需要通过网络与 Hadoop 集群中的所有服务器进行通信。

HDFS 是一种主 / 从（head/worker）架构，由两个基本组件构成：NameNode（主）和 DataNode（从）。附加的组件包括 JournalNode、HttpFS 和 NFS 网关。

NameNode

NameNode 负责跟踪 HDFS 中所有与文件相关的元数据，例如文件名、块位置、文件许可和副本。从安全角度需要知悉的是，那些读写文件的 HDFS 客户端总是与 NameNode 通信的。此外，NameNode 为整个 Hadoop 生态系统提供了一些重要的安全功能，这些内容将在之后进行阐述。

DataNode

DataNode 负责在 HDFS 中对数据块进行存储和读取。NameNode 会告诉正在读取某文件的 HDFS 客户端，集群中的哪个 DataNode 拥有客户端请求的数据。向 HDFS 写文件时，客户端向一个由 NameNode 指定的 DataNode 写入数据块。DataNode 由此建立一个通向其他 DataNode 的写管道，从而完成基于所需复制因子的写操作。

JournalNode

JournalNode 是 HDFS 中的一种特殊组件。HDFS 被配置为高可用性（**HA**）时， JournalNode 会接管 NameNode 写入 HDFS 元数据信息的职责。集群通常具有奇数个 JournalNode（3 个或 5 个），用以确保能满足对仲裁资源的要求（不能出现平局）。例如，如果要向 HDFS 写入一个新文件，该文件的元数据会被写入每一个 JournalNode。当多数 JournalNode 成功写入该信息，发生的改变（写入行为）会被认为是有效的。 HDFS 客户端和 DataNode 不会直接与 JournalNode 互动。

HttpFS

HttpFS 是 HDFS 中负责向客户端提供与 NameNode 和

DataNode 连接代理的组件。该代理是一种 REST API，它允许客户端——无需与 HDFS 中的其他组件建立直连——通过代理使用 HDFS。HttpFS 在某些集群的架构中将会是关键的组件，稍后阐述。

NFS 网关

顾名思义，NFS 网关允许客户端像使用 NFS 文件系统那样使用 HDFS。NFS 网关实质上是一个帮助客户端和底层 HDFS 集群间进行 NFS 协议通信的守护进程。与 HttpFS 很相似，NFS 网关位于 HDFS 和客户端之间，因此能够为特定的集群架构提供安全边界。

KMS

Hadoop 关键管理服务，亦作 **KMS**，在 HDFS 的静态透明加密功能中扮演着重要角色。它的功能是作为 HDFS 客户端、NameNode 和关键服务之间的中间人，处理加密操作，如解密数据加密密钥和管理加密区密钥等。这部分内容将在第 9 章进行详细阐述。

1.3.2　Apache YARN

随着 Hadoop 的发展，MapReduce 即便拥有强大功能，也明显不再能够满足新的用例需求。许多问题不能利用 MapReduce 的编程模型轻易解决，人们需要一种能够适应新处理模型的更通用的框架。Apache YARN 就提供了这种能力。其他处理框架如 Impala、Spark 等，也将 YARN 用作它们的资源管理框架。虽说 YARN 提供的是更通用的资源管理框架，但 MapReduce 依然是运行在它之上的标准应用。这种运行在 YARN 上的 MapReduce 被认为是 v2 版，简称 MR2。YARN 的架构由以下几部分组成。

ResourceManager

ResourceManager 守护进程负责处理应用提交请求、分配 ApplicationMaster 任务，以及执行资源管理策略。

JobHistory Server

顾名思义，JobHistory Server 负责跟踪在 YARN 上运行的所有作业历史，包括运行时间、任务数量、写入 HDFS 的数据量等作业的度量。

NodeManager

NodeManager 守护进程负责执行 YARN 容器（**container**）中作业的各个任务，每个任务由虚拟内核（CPU 资源）和 RAM 资源组成，并可以按照各自需要申请一定数量的虚拟内核和内存，其最小值、最大值和增幅由 ResourceManager 定义。每个任务作为独立的进程在各自的 JVM 中运行。NodeManager 的一个重要作用是，启动名为 **ApplicationMaster** 的特殊任务，该任务负责管理指定应用的所有任务状态。YARN 将资源管理和任务管理区分开，这样每个作业都能执行各自的 ApplicationMaster，从而能够在大型集群中更好地扩展 YARN 应用。

1.3.3　Apache MapReduce

MapReduce 是与 HDFS 相对应的数据处理部分，提供最基本的数据批处理机制。在 YARN 上执行的 MapReduce 常被称为 MapReduce2 或 MR2，人们以此区分基于 YARN 的 MapReduce 和独立的 MapReduce 框架（后者被追加命名为 MR1）。MapReduce 作业由客户端提交给 MapReduce 框架，然后在 HDFS 中的一个数据子集（通常是一个特定目录）上执行。MapReduce 自身也是一个编程模型，允许多个服务器并行、独立地处理数据块。虽然 Hadoop 开发者需要了解 MapReduce 复杂的工作原理，但安全架构师多半不需要了解这些。后者需要知道的是，客户端提交作业到 MapReduce 框架，并且从那之后，由 MapReduce 框架负责客户代码在集群之间的分发和执行。客户端不需要与集群的任何节点进行交互以执行作业，而是作业本身会申请运行一定数量的任务（**task**）完成其工作。依据 MapReduce 框架的调度算法，每个任务会被分配到一个节点运行。

> “MapReduce 框架在指定服务器上建立的各个任务以什么用户身份执行”，这取决于 Kerberos 是否开启。如果 Kerberos 没有开启，每个任务都会以 `mapred` 系统用户的身份执行。Kerberos 开启时，各个任务会以提交 MapReduce 作业的用户身份执行。但即便开启了 Kerberos，当另外一个组件或工具正在提交 MapReduce 作业时，可能无法立刻看出哪个用户在执行该作业下的 MapReduce 任务。涉及 Hive 身份模拟的相关细节讨论参见 5.2.4 节。

与 HDFS 类似，MapReduce 也是一个主 / 从式架构，包括两个主要部分。

JobTracker（主）

客户端向 MapReduce 框架提交作业时，与客户端进行通信的其实是 JobTracker。 JobTracker 负责处理客户端的作业提交，并且决定作业应该如何按照设定好的方式运行，例如作业需要多少任务、哪个 TaskTracker 负责哪个任务等。另外，JobTracker 还负责处理安全和运行相关的功能，例如作业队列、调度池，以及用于验证的访问控制表等。最后，JobTracker 还负责处理作业的度量和其他相关信息，这些信息在作业执行的过程中会被各种 TaskTracker 发送给 JobTracker。JobTracker 包括资源管理和任务管理两部分，这两部分在 MR2 中被 ResourceManager 和 ApplicationMaster 分隔开。

TaskTracker（从）

TaskTracker 负责执行一个 MapReduce 作业中的给定任务。TaskTracker 从 JobTracker 接收要运行的任务，并为每个运行的任务生成独立的 JVM 进程。TaskTracker 既执行 map 任务，也执行 reduce 任务，具体可以并发执行多少个任务取决于 MapReduce 的配置。从安全角度看，其要点是，JobTracker 决定运行什么任务以及任务运行在哪些 TaskTracker 上。通常的作业执行过程中，客户端既不能控制任务如何分配，也不与 TaskTracker 进行任何通信。

关于 MapReduce 的一个要点是，其他的 Hadoop 生态系统组件都是建立在 MapReduce 之上的框架和库。这意味着，虽然 MapReduce 负责实际的数据处理，但这些框架和库将 MapReduce 的作业执行从客户端中抽象出来。Hive、Pig、Sqoop 组件都这样使用 MapReduce。

> Hadoop 中，用户审计的一个重要部分就是理解 MapReduce 作业是如何提交的，具体细节参 5.2.3 节“块访问令牌”部分。从安全角度看，同样是使用 MapReduce，用户提交自己的 Java MapReduce 代码与使用 Sqoop 从 RDBMS 导入数据或在 Hive 中执行 SQL 查询相比，是完全不同的活动。

1.3.4　Apache Hive

Apache Hive 项目最早由 Facebook 发起。Facebook 看到了 MapReduce 的实用性，但也发现，分析人员缺乏 Java 编程技能，导致 MapReduce 框架的适用性大打折扣。由于大部分 Facebook 分析人员具备 SQL 技能，因此 Facebook 发起了 Hive 项目，使用 MapReduce 作为执行引擎的 SQL 抽象层。Hive 架构由以下几部分组成。

Metastore 数据库

Metastore 数据库是一个包含所有 Hive 元数据（例如库、表、列、数据类型等信息）的关系型数据库。这种信息在访问 HDFS 时将数据结构化，也被称为“读模式”（**schema on read**）。

Metastore Server

Hive Metastore Server 是处于 Hive 客户端和 Metastore 数据库之间的一个守护进程，它提供了一个安全层，防止客户端拥有 HiveMetastore 的数据库凭证。

HiveServer2

HiveServer2 是客户端使用 Hive 的主入口。HiveServer2 兼容 JDBC 和 ODBC 客户端，因此被很多客户端工具和其他第三方应用使用。

HCatalog

Hcatalog 是一系列帮助非 Hive 框架访问 Hive 元数据的库。例如 Pig 用户可以用 Hcatalog 读取与 HDFS 中给定目录相关的模式信息。WebHCat Server 是一个向客户端提供 REST 接口的守护进程，使客户端能够访问 HCatalog 的 API。

如果想全面了解 Hive，可以阅读《Hive 编程指南》。

1.3.5　Cloudera Impala

Cloudera Impala 是一个大规模并行处理（Massive Parallel Processing，MPP）框架，专门为了分析型 SQL（Analytic SQL）而建立。Impala 从 HDFS 读取数据，并利用 Hive Metastore 解析数据结构和数据格式。Impala 架构由以下几部分组成。

Impala 守护进程（impalad）

Impala 守护进程负责数据处理中的所有繁重任务，这些服务与 HDFS DataNode 相配合，以优化本地读取。

StateStore

StateStore 服务保存所有正在运行的 Impala 守护进程的状态信

息。它监视 Impala 守护进程的状态是开启还是关闭，并向所有服务广播该状态。StateStore 并不是 Impala 架构中必需的组件，但它可以在一个或多个服务宕掉时提供更快的容错能力。

Catalog Server

Catalog Server 是从 Impala 到 Hive Metastore 的入口。该进程负责从 Hive Metastore 获取元数据，以及对 Impala 客户端修改过的元数据进行同步。有一个独立的 Catalog Server 既能够降低 Hive Metastore Server 的负载，也能够为 Impala 额外提供速度优化。

> 接触 Hadoop 生态系统的新用户经常会问，Hive 和 Impala 都提供对 HDFS 数据的 SQL 访问能力，那它们的区别是什么？ Hive 让熟悉 SQL 的人在不需要对 MapReduce 进行任何了解的前提下访问 HDFS 数据，它的设计目标是对 MapReduce 的内部结构进行抽象，从而使得访问 HDFS 数据更加容易，广泛应用于批处理和 ETL（Extract-Transform-Load，即数据的抽取、转换、加载）工作；而 Impala 则是自底向上设计的快速分析型处理引擎，支持即席查询和商业智能（business intelligence，BI）工具。Hive 和 Impala 都很实用，应当视为互补组件。

如果想全面了解 Impala，请参考 *Getting Started with Impala*。

1.3.6　Apache Sentry

Sentry 组件（孵化中）为其他生态系统组件（如 Hive、Impala 等）提供细粒度角色访问控制（role-based access control，RBAC）。虽然各个组件可能会有自己的验证机制，但 Sentry 在各个组件之间提供了一个支持强制集中策略的统一验证机制。Sentry 是 Hadoop 安全的一个重要组件，我们将用整整一章介绍它（第 7 章）。Sentry 由以下几部分组成。

Sentry Server

Sentry server 是一个守护进程，方便查找其他 Hadoop 生态系统组件产生的策略。Sentry 的客户端组件会根据 Sentry 置入的策略继承验证决策。

策略库

Sentry 策略库存储所有验证策略。Sentry Server 根据策略库决定一个用户的特定操作是否被允许。具体来说就是，Sentry Server 会为用户寻找一个匹配的策略，以授予其对资源的访问权限。早期版本的 Sentry 中，策略库仅是一个包含所有策略的文本文件。 Sentry 和策略库的发展会在第 7 章详细讨论。

1.3.7 Apache HBase

Apache HBase 是分布式键 / 值存储，由谷歌的 BigTable 论文“BigTable: A Distributed Storage System for Structured Data”启发而来。HBase 通常使用 HDFS 作为数据的底层存储层，结合本书要讨论的内容，我们假定书中提到的 HBase 均为这种情况。**HBase** 表被划分到不同区域，这些区域根据行键（**row key**，键值的索引部分）进行分隔。行 ID 是顺序排列的，所以一个指定的区域具有一系列顺序的行键。区域由 **RegionServer** 存放，客户端使用键值从 RegionServer 请求数据。键值包括几个部分：行键、列族（**column family**）、列限定符（**column qualifier**）和时间戳（**timestamp**）。这些部分组合起来能够唯一地确定存储在表中的一个值。

客户端访问 HBase 时，首先通过搜索 `hbase:meta` 表查找负责存放特定范围行键的 RegionServer。找到正确的 RegionServer 后，客户端会直接向该 RegionServer 发起读写请求，而不是通过 Master 发起。客户端会缓存 RegionServer 的区域映射，以避免重复的查询过程。存放 `hbase:meta` 表的服务器位置可以在 ZooKeeper 中查到。HBase 包括以下几部分。

Master

正如之前提到的，HBase Master 守护进程负责管理哪个区域由哪个 RegionServer 存放。如果某个 RegionServer 宕掉，HBaseMaster 也要负责将它存放的区域重新分配给其他 RegionServer。多个 HBaseMaster 可以并行运行，在任意时刻，它们会通过 ZooKeeper 选择一个 HBaseMaster 保持活跃。

RegionServer

RegionServer 负责为指定的 HBase 表提供区域。区域是一系列特定范围的键值，这些键值既可以通过 HBase 控制台人工确定，也可以由 HBase 根据曾经存入表的键值自动确定。HBase 的一个目标是平均分配键值空间，使每个 RegionServer 具有同等的提供数据的责

任。每个 RegionServer 通常存放着多个区域。

REST Server

HBase REST Server 提供执行 HBase 指令的 REST API。和 Hadoop 生态系统中的许多其他项目一样，默认的 HBase API 是用 Java API 提供的。REST API 通常被用作一种独立于编程语言的接口，使客户端能够使用任何他们想用的编程语言。

Thrift Server

除 REST Server 之外，HBase 还有一个 Thrift Server，它为客户端提供了另外一种可用的 API 接口。

要想了解更多 HBase 架构和最适用的用例，推荐阅读《HBase 权威指南》。

1.3.8　Apache Accumulo

Apache Accumulo（http://accumulo.apache.org）是一个排序分布式的键 / 值存储，一个健壮的、可扩展的、高性能的存储和检索系统。与 HBase 类似，Accumulo 起初也是基于 Google BigTable 设计的，但它建立在 Apache Hadoop 生态系统（尤其是 HDFS、ZooKeeper 和 Apache Thrift）之上。Accumulo 使用的数据模型和 HBase 大体一致，每个 Accumulo 表被分成一个或多个块，每个块包含基本相同数量的记录，这些记录根据行 ID 分配。每个记录也有一个由多部分组成的列键，包括列族、列限定符和一个可见性标签（visibility label）。可见性标签是 Accumulo 与原始 BigTable 设计最大的不同点之一，它增加了实现单元级安全的能力（我们将会在第 6 章详细讨论）。最后，每个记录也包含一个时间戳，使用户可以存储多个版本的记录。如果没有时间戳，这些不同版本的记录的键值将相同。综上，行 ID、列和时间戳组成了确定一个特定记录的键值。

块的分配以分割行 ID 集合的方式进行，分割点在数据被插入表时自动计算。每个块由一个负责在该块中提供数据读写服务的 TabletServer 存放，每个 TabletServer 可以存放来自一个或多个表的多个块，所以块是系统中的分配单元。

客户端第一次访问 Accumulo 时，会查找存放 `accumulo.root` 表的 TabletServer 位置，`accumulo.root` 表保存着 `accumulo.meta` 如何被分割到块的信息。客户端会直接与存放

`accumulo.root` 的 TabletServer 通信，然后再与存放 `accumulo.meta` 表数据块的 TabletServer 通信。由于这些表中（尤其是 `accumulo.root` 中）数据的改变频率与其他数据相比相对较少，所以客户端会保存一个从这些表读取的数据块位置的缓存，以避免读写管道中的瓶颈。一旦客户端知道了它正在读 / 写的行 ID 所属的数据块位置，它就会直接与对应的 TabletServer 通信。客户端在任何时候都不会与 Master 通信，这一点在很大程度上保证了可扩展性。总体来说，Accumulo 包含以下几部分。

Master

Accumulo Master 负责协调分配数据块到 TabletServer，保证每个块被存放在唯一的 TabletServer 中，并对 TableServer 失效等事件进行响应。它还负责表的管理性更改以及协调启动、关闭和写前日志的恢复。多个 Master 可以同时运行，它们会选出一个领导者，从而使每个时刻只有一个 Master 处于活跃状态。

TabletServer

TabletServer 负责处理对 Accumulo 集群中数据块子集的所有读写请求。就写操作而言，它负责周期性地将记录写入写前日志，并将内存中的记录写入磁盘。恢复过程中， TabletServer 将写前日志中的记录回放到要恢复的数据块。

GarbageCollector

GarbageCollector 周期性地删除 Accumulo 进程不再需要的文件。多个 GarbageCollectors 可以同时运行，它们会选择一个领导者，从而使每个时刻只有一个 GarbageCollector 处于活跃状态。

Tracer

Tracer 监控集群中使用 Accumulo 分布式定时 API 的其他部分，并将这些数据写入一个 Accumulo 表，以备后用。多个 Tracer 可以同时运行，它们会平均分配负载。

Monitor

Monitor 是一个用于监控 Accumulo 集群状态的 Web 应用，展示了记录数量、缓存命中率 / 未命中率等关键度量值，以及扫描率等数据表信息。Monitor 还充当了日志转发节点的角色，使用户可以通过单一接口诊断错误和警告信息。

1.3.9 Apache Solr

Apache Solr 项目又称 **SolrCloud**，能够对分散于不同物理服务器的文档（作为更大数据集合的一部分）进行搜索和检索。搜索是大数据的经典用例之一，也是任何人访问互联网都会用到的最常见的功能。Solr 建立在 Apache Lucene 项目之上，由 Lucene 实际负责大部分的检索和搜索工作。Solr 提供逐级导航（faceted navigation）、高速缓存、命中字高亮显示、管理接口等企业级功能，以对这些能力进行扩展。

Solr 是一个单独的组件，即 Server。一个部署环境中可以有多个 Solr Server，并通过 SolrCloud 提供的分片机制线性扩展。SolrCloud 还提供了自我复制功能，以应对分布式环境中的故障。

1.3.10 Apache Oozie

Apache Oozie 是一个 Hadoop 工作流和业务流管理系统。它可以建立包含多个动作的工作流，每个动作可以使用 Hadoop 生态系统中的不同组件。例如，一个 Oozie 工作流可以首先执行一个 Sqoop 导入动作，将数据移入 HDFS；然后执行一个 Pig 脚本，对数据进行转换；之后再用一个 Hive 脚本建立元数据结构。Oozie 支持多个复杂的工作流，例如使用分叉（fork）和连接（join）动作可以并行执行多个步骤，或者执行依赖于多个步骤完成之后才能继续的其他步骤。Oozie 工作流可以根据不同输入条件，按照周期性的计划运行，例如在特定时间运行，或者等待 HDFS 中某个路径存在后运行。

Oozie 只有一个服务组件，该服务负责处理客户端的工作流提交、管理工作流的执行以及状态报告。

1.3.11 Apache ZooKeeper

Apache ZooKeeper 分布式协同服务允许分布式系统同步存储和读取小量数据，常用于存储通用配置信息。此外，ZooKeeper 还经常用于同步 Hadoop 系统中的高可用性服务，例如元数据节点 HA 和资源管理 HA。

ZooKeeper 本身也是一个分布式系统，它通过奇数个 ZooKeeper **Ensemble** 服务器的仲裁，确认下发的事务。ZooKeeper 只有一个组件，即 ZooKeeper Server。

1.3.12　Apache Flume

Apache Flume 是一个基于事件的数据导入工具，主要用于将数据导入 Hadoop，也可完全独立使用。Flume 的意思是槽，顾名思义，其作用是将事件日志导入 HDFS。Flume 框架主要由三部分组成：数据源（source）、数据阱（sink）和通道（channel）。

Flume 的数据源定义了应当如何从上游提供者读取数据。读取流程可能包括 syslog 服务器、JMS 队列，甚至轮询某个 Linux 目录等。Flume 数据阱定义了数据应当如何被写入下游。通用的数据阱包括一个 HDSF 阱节点和一个 HBase 阱节点。最后，Flume 通道定义了数据在源和阱之间是如何被存储的。最基本的两种通道是内存通道和文件通道。内存通道通过牺牲一定的可靠性实现高速传输，文件通道则通过牺牲一定的传输速度提供较高的可靠性。

Flume 中只有一个组件，即 Flume **Agent**。Agent 包含数据源、数据阱和通道的代码。Flume 架构的一个重要特性是：Flume Agent 之间能够互联，即某个 Agent 的数据阱能够与另一个 Agent 的数据源连通。这种情况下的一个通用接口是 Avro 源和阱。Flume 数据导入和安全将在第 10 章介绍，也可翻阅 *Using Flume*（http://shop.oreilly.com/product/0636920030348.do）。

1.3.13　Apache Sqoop

Apache Sqoop 提供了在传统 RDBMS 以及其他数据源（如 FTP 服务器）中批量导入 / 导出数据的功能。Sqoop 自身通过提交 map-only 的 MapReduce 任务，与 RDBMS 进行并行交互。Sqoop 既可作为初始化 Hadoop 集群数据的一种简单机制，也可以是进行常规数据导入 / 导出工作的一种工具。目前有两种不同版本的 Sqoop：Sqoop1 和 Sqoop2。本书重点讲解 Sqoop1。 Sqoop2 在本书写作之时尚未完善，并且缺乏一些基本的安全特性，如 Kerberos 身份验证。

Sqoop1 是由使用 sqoop 程序的命令行衍生的一系列客户端库。这些客户端库负责实际的 MapReduce 作业提交，即将作业提交到合适的框架（例如传统的 MapReduce 或者基于 YARN 的 MapReduce2）。Sqoop 的详细讨论参见第 10 章，也可翻阅 *Apache Sqoop Cookbook*（http://shop.oreilly.com/product/0636920029519.do）。

1.3.14　Cloudera Hue

Cloudera Hue 是一个 Web 应用，将许多 Hadoop 生态系统组件友好地暴露给用户。Hue 允许用户无需熟悉 Linux 或者各种命令行接口，就可轻松访问 Hadoop 集群。Hue 拥有一些不同的安全控制策略，这些将在第 12 章讲到。Hue 由如下组件组成。

Hue Server

Hue Server 是 Hue 的主要组件，它实质上是一个向用户提供网页内容的 Web 服务。用户第一次登录时需要进行验证，之后，终端用户执行的动作实质上是由 **Hue** 代理用户完成的。这就是第 5 章将介绍的身份模拟（**impersonation**）。

Kerberos Ticket Renewer（Kerberos 票据更新器）

顾名思义，该组件负责周期性地更新 Kerberos 的票据授予票据（**ticket-granting ticket ，TGT**），在 Hadoop 集群启用 Kerboros 的前提下，该票据被 Hue 用于与 Hadoop 集群进行交互（第 4 章对 Kerberos 有详细阐述）。

1.4　小结

本章介绍了一些常见的安全术语，这些安全术语是本书其他内容的基础。阅读本章的一个关键收获是，读者能够认识到，Hadoop 安全并非那么难以理解。传统可靠的安全准则（如 CIA 和 AAA）在 Hadoop 环境中同样适用，这些会在后面的章节中详细讨论。最后，我们回顾了很多 Hadoop 生态系统项目（及其组件），并逐个了解它们的作用，并大致理解安全是如何应用的。

下一章我们会讨论保护分布式系统的安全。你会发现，很多 Hadoop 上的安全威胁和缓减方法也同样适用于分布式系统。

第一部分　安全架构

第 2 章　保护分布式系统

第 1 章介绍了安全计算的一些关键原理，本章将进一步了解思考分布式系统安全时面临的一些有趣挑战。接下来你将看到，分布式大大增加了系统的潜在威胁，从而也增加了为帮助减轻这些威胁而必需的安全评估的复杂度。我们将用一个真实的案例说明安全需求如何随着系统的分布式程度而增加。

以银行为例。很多年前，对普通人而言，银行业务就是开车去当地银行，找个出纳员，然后亲自进行交易。而银行的安全措施可能包括检查客户的身份和账号，以及验证用户要求的操作可以执行，比如在取现时确保账户内有足够的钱。

多年来，银行越来越大。小镇上的银行变成了一个更大银行的支行，因此，你不仅可以在附近的这个银行支行办理业务，也可以在其他地方的该银行支行办理。这时，保护资产所必需的安全措施也有所增加，因为需要保护的不再是一个地点。同时，更多的银行出纳员也需要进行妥善的培训。

更进一步，银行终于使用上了 ATM 机，客户因此不需要去支行就能取现。你可能会想到，与以前银行业务在人与人之间进行交互的情况相比，这时需要更多安全措施保护银行。随后，银行之间开始相互联网，从而允许一家银行的客户使用另一家银行的 ATM 机。这时，银行需要相互之间建立安全控制措施，以确保这种互联不会造成安全损失。最后，互联网运动引入了通过网站甚至移动设备进行网上银行业务的能力，这又大大增加了银行的潜在威胁，以及对安全控制措施的需求。

如你所见，刚开始，保护小镇上的小银行只是一个简单的安全任务。随着银行在数十年来变得越来越分散和互联，难度增加了好几个数量级。虽然这个例子看起来很浅显，但它表达了如何为分布到数十、数百甚至数千台机器上的系统设计安全框架的问题。这是个不小的任务，为了使这个任务不那么可怕，我们可以对它进行拆解。首先，从理解威胁开始。

2.1　威胁种类

要想为分布式系统设计健壮的安全架构，一个关键的要素是理解可能

出现的威胁，并能够对威胁进行分类，从而更好地理解要降低这些威胁需要何种安全机制。本节将回顾一些需要关注的、常见的威胁类别。

2.1.1 非授权访问 / 伪装

非授权访问是最常见的威胁类别之一。这种情况出现在某个人应当被拒绝访问时，反而成功进入系统。非授权访问的一种常见方法是伪装攻击。伪装的概念是，一个非法用户冒充为一个合法用户，从而获取访问权。你可能会问："非法用户是如何冒充为合法用户的？"最可能的答案是，攻击者获取了一个合法用户的用户名和口令。

伪装攻击自从互联网时代开始就特别突出，尤其是针对分布式系统的伪装攻击。攻击者有很多种方法获取合法的用户名和口令，比如尝试常见的单词和词组，或者了解与合法用户相关的、可能会被用作密码的单词。比如攻击者想获取一个社交媒体网站的登录凭证，那么他可能会从用户的公开帖子中收集关键字，并组成可供尝试的密码列表。举个例子，如果攻击者关注身在纽约、将"棒球"列为兴趣爱好的用户，那么他可能会尝试密码 `yankees`（yankees 是纽约扬基队的队名）。

如果非法用户成功实施了伪装攻击，安全管理员如何才能得知呢？毕竟攻击者登录时使用的是合法用户的凭证，从分布式系统的角度看，这不是正常行为吗？不见得。通常情况下，可以查看审计日志中的尝试登录行为以发现伪装攻击。如果攻击者使用可能的密码列表对某个用户账号进行尝试，那么不成功的尝试行为应当出现在审计日志文件中。如果看到有针对某个用户的大量失败的尝试登录行为，那么通常可归因于攻击。一个合法用户有可能输错或者忘记密码，从而导致少量失败的尝试登录事件，但比如 20 次连续的登录失败就应该是异常的了。

另外一种发现伪装攻击的常见途径是，从网络层面看，登录尝试的来源是哪里。按照 IP 地址分析登录尝试行为是发现是否有人在尝试伪装攻击的好方法。尝试登录的客户端 IP 地址是否与期望相符，比如是来自公司 IP 地址的一个已知子网，还是来自世界另一端的某个国家？还有，登录尝试行为发生在什么时间？ Alice 是在凌晨 3 点还是在正常的工作时间尝试登录系统的？

非授权访问的另外一种形式是攻击者利用系统的漏洞，从而不需要提供合法凭证就能进入系统。我们将在 2.3 节讨论漏洞。

2.1.2 内在威胁

内在威胁可以说是最具破坏性的威胁类型。顾名思义，攻击者来自业务内部，并且是一个正常用户。内在威胁可以包括员工、顾问、承包商。内在威胁之所以可怕是因为，攻击者已经具有系统的内部访问权限。攻击者可以使用合法凭证登录并获得系统授权，以执行特定功能，通过途中的许多安全检查（因为攻击者被认为应当被授予访问权限）。这种威胁可以导致对系统的肆无忌惮的攻击，或者诸如攻击者利用自己的权限把敏感数据泄漏给非授权用户等更为巧妙的攻击。

本书中，你将会找到确保用户只访问他们所需的数据和服务的安全特性。对付内在威胁需要有效的审计实践（参见第 8 章）。除了能够帮助对抗内在威胁的技术工具之外，还需要建立业务策略强制实施合适的审计，对事件响应的流程也需要明确。虽然此处并未涉及如何建立这种策略的最佳实践，但这些策略对本章描述的所有威胁类型而言都是必要的。

2.1.3 拒绝服务

拒绝服务（**Denial of Service**，**DoS**）是指，一个服务对于一个或多个用户而言不可用。“服务”在这里是一个概括性词汇，包括数据访问、处理能力以及相关系统的可用性。拒绝服务的发生可以来自各种各样的攻击方式。在互联网时代，一种常见的攻击方式是，简单地使用大量网络流量压垮系统。这需要使用很多机器并行进行，因此使得这种攻击变为分布式拒绝服务（**Distributed Denial of Service**，**DDoS**）。当系统被过多需要处理的请求轰炸，就会开始在某些方面产生故障，小到丢弃其他合法请求，大到系统完全失效。

虽然分布式系统通常受益于其具有的某些容错能力，但 DoS 攻击仍然是可能的。例如，如果一个分布式系统包含 50 台服务器，攻击者可能很难扰乱全部 50 台设备的服务。但如果该分布式系统仅仅部署在少量网络设备后面，例如 1 台网络防火墙和 1 台接入交换机呢？攻击者可以将分布式系统的网关——而不是分布式系统本身——作为目标，通过这种方法达到自己的目的。这一点很重要，我们将在第 3 章——讲述建立集群周边的网络边界——对其进行讨论。

2.1.4 数据威胁

数据是分布式系统中最重要的部分。没有了数据，一个分布式系统就只是数据中心里一堆“嗡嗡”响的空闲服务器，只能徒增电力和冷却费

用。正因为数据如此重要，它也是安全攻击的重点。数据威胁在一个分布式系统中出现在多个地方。首先，数据必须以安全的方式存储，以防止非授权的查看、篡改或删除。其次，数据在传输过程中也必须得到保护，因为分布式系统毕竟是分布式的。通过网络传递数据可能会被 DoS 攻击等破坏性的攻击威胁，也可能会被一些更为被动的攻击方式威胁，例如攻击者在通信方不知道的情况下抓取网络通信数据。第 1 章讨论了 CIA 模型及其组件，CIA 模型在根本上就是关于降低数据威胁的。

2.2 威胁和风险评估

你可能不是第一次听到上一节中对威胁类别的叙述，在理解这些威胁类别的基础上，对实际的分布式系统进行风险评估是很重要的。例如，拒绝服务攻击更有可能发生在直接连接到互联网的系统上，而对于那些不能从外网访问的系统，如公司内网中的系统，这种攻击发生的风险就低得多。注意，是风险低而不是没有风险，这是个很重要的区别。

评估对分布式系统的威胁需要进一步了解两个要素：用户和环境。理解这两个要素后，评估风险就更容易了。

2.2.1 用户评估

了解分布式系统会暴露给什么用户是很重要的。这明显包括访问系统的用户和直接与系统接口交互的用户，也包括可能出现在别处但不会直接访问系统的用户。理解这种环境下的用户有助于更好地进行风险评估。首先根据分布式系统（如 Hadoop）的用户工作对其进行分类。用户是做什么的？他们是业务情报分析员吗？或者是开发者？还是风险分析员？还是安全审计员？抑或是数据质量分析员？

一旦用户根据业务功能被分成不同的组，就可以开始识别不同组用户使用分布式系统的访问模式和工具。例如，如果分布式系统的用户都是开发者，那么可以假定他们需要系统中节点的控制台（shell）访问、调试作业的日志文件和开发工具。另一方面，业务情报分析人员可能不需要以上任何一种，而需要一套代替用户与分布式系统交互的分析工具。

还有间接访问系统的用户。这些用户不需要系统数据或处理资源的访问权限，但他们仍会与系统进行交互。这种交互作为某些功能的一部分，比如一些系统维护、健康监控、用户审计的支持功能。这种类型

的用户需要被考虑到总体的安全模型中。

2.2.2 环境评估

为了对分布式系统进行风险评估，还需要了解系统所在环境。这通常意味着要对本系统与其他逻辑系统相关的、物理环境相关的操作环境都进行评估。第 3 章将介绍 Hadoop 专有的相关内容。

如前所述，环境评估的关键准则之一是，判断分布式系统能否通过互联网访问。如果能，则说明会有大量威胁可能发生，如 DoS 攻击、漏洞利用和病毒。连接至互联网的分布式系统需要长期监测，就像应用软件需要定期打补丁、安全软件需要定期升级一样。

对环境评估的另一准则是，要了解组成分布式系统的服务器的物理位置。它们位于自己公司的数据中心吗？还是位于第三方管理的数据中心？抑或是部署在公有云设施上？了解这些问题的答案会有助于开始建立安全评估的框架。例如，若分布式系统搭建在一个公有云上，那么这些威胁会立刻显现：基础架构不由自己的公司拥有，因此用户不清楚谁对这些机器拥有直接访问权，这使得主机服务提供商也被囊括进内部威胁之中。同样，公有云的使用回避了用户如何连接至分布式系统，以及数据如何流入 / 流出系统的问题。一个开放网络到一个共享的公有云之间的通信受到的威胁级别要比公司内部数据中心之间通信的威胁级别高得多。

这样说的重点不是要让大家认为，公有云不好而公司的数据中心好；而是要告诉读者，对于分布式系统已知威胁的级别会由于其存在形式的不同而不同。若不考虑环境因素，那么保护分布式系统的关键在于从多种层面降低风险，这些内容将会在下一节讨论。

2.3 漏洞

虽然漏洞是一个独立话题，但也与威胁和风险相关。分布式系统中，漏洞以多种不同形态存在。一个常见的存在漏洞的地方是软件本身，所有软件都有漏洞。这听起来像是个武断的结论，但事实是，没有软件是 100% 安全的。

那么，到底什么是软件漏洞呢？简言之，漏洞就是一段代码，这段代码容易受到未被考虑的错误或者无效条件的影响。下面看一个简单的实例。一个软件含有一个输入密码的界面，该界面允许用户修改密码（假设软件的内在逻辑允许至多 16 个字符长度的密码）。如果新密

码的输入框长度被错误地限制为至多 8 个字符，导致用户选择的新密码被错误截断，那么会发生什么呢？这可能导致用户实际上设定的密码比他们认为的要短，或者导致更糟的情况：复杂度较低的密码更容易被攻击者猜出。

当然，软件漏洞不是分布式系统容易遭受的唯一漏洞，其他漏洞还与分布式系统依赖的网络架构设施有关。例如，多年前有个漏洞，允许攻击者向网络的广播地址发送 ping 命令，这导致网络中的每一台主机都会回复 ping 命令。攻击者构造 ping 的请求，将发起请求的源 IP 设定为网络中其试图攻击的某个主机 IP。这样做的结果就是，目标主机被网络中的通信淹没，从而导致故障。这种攻击手段通常称为“死亡之 **ping**”（**ping of death**）。该风险现在已经缓解，但重点是，虽然该漏洞与网络中机器的软件没有关系，但在网络硬件供应商修复该漏洞之前，攻击者依然能够利用这个漏洞破坏网络中的特定主机。

软件补丁通常用于修复被发现的漏洞，因此，定期按计划更新补丁包是分布式系统软件栈中每个管理员的标准操作规程中不可缺少的部分。正如“死亡之 ping”的例子展示的那样，补丁的范围也应包括交换机、路由器、其他网络设备、硬盘控制器以及服务器 BIOS 的固件。

2.4 深度防御

安全管理员面临的众多挑战之一是，如何缓解本章提到的所有威胁和漏洞。研究多种威胁时，人们很容易发现，没有一个“放之四海而皆准”的方法能够有效消除这些威胁。为了防御这些威胁，提供一个可接受的安全等级，需要部署很多安全控制策略——并且它们必须协同工作。这种同时部署多种安全控制策略和保护措施的方法就是深度防御。

回顾历史，深度防御并未被常常遵守。“安全”通常指边界安全，此概念中，安全控制策略仅仅存在于受保护目标的外部或边界。一个经典的例证便是想象一个堡垒，其周边围有很厚的高墙。一般的思维定势会认为，只要高墙不倒，堡垒就是安全的。如果高墙被攻陷，堡垒里的人就危险了。如今，情况好了很多。

现在，深度防御存在于我们的日常生活。以去杂货店为例。杂货店有一扇上了锁的门，只有日间正常营业时间才会打开。同时，店里也有警报系统，入侵者在非营业时间非法闯入时会触发。在正常的营业时间，购物者的行为被遍布商店的安全摄像头监控着。最后，商店雇员也受到专门培训，以注意那些形迹可疑的客户。

所有这些安全措施的部署都是为了针对多种不同威胁，例如入室盗窃、伪装成顾客的扒手，以及抢劫等，以保护商店。如果杂货店仅仅依赖于“城堡高墙”的策略加强门和锁，将会对大部分威胁束手无策。深度防御很重要，因为任何单一的安全策略均无法缓解商店面临的所有风险。对于分布式系统也一样，在很多地方能够部署独立的安全策略。例如在边界配置网络防火墙，严格限制对数据的访问，或者限制对服务器的访问等，但同时使用所有这些策略才能降低攻击的成功率。

2.5　小结

本章通过分析威胁种类和漏洞分解了分布式系统的安全体系，并且展示了使用深度防御的安全架构能够将安全风险减少到最低。我们还讨论了内部威胁，以及为什么不能在设计安全架构时忽视它。

第 3 章将从建立一个健壮的系统架构开始，重点探讨 Hadoop 的保护措施。

第 3 章　系统架构

在第 2 章我们看到，随着单个独立的系统变为完全分布式的网络系统，其安全形势也发生了改变。我们能清楚地看到，保护一个 Hadoop 集群中成百上千的服务器有多么困难。本章将把集群分解为若干组件，这些组件可作为整体安全策略的一部分进行单独保护，从而深入探讨这个艰巨的任务。Hadoop 集群大致可分为两个主要部分：网络和主机。在此之前，先看看 Hadoop 集群的运行环境。

3.1　运行环境

在 Hadoop 早期，集群可能意味着一堆原本被用于其他用途，而现在被用来尝试新技术的机器，甚至可能是用一些陈旧的桌面级机器加上一些接入交换机组成的。多年来，事情发生了巨大的改变。在房间角落堆一些机器的日子已经被"Hadoop 集群是企业中的头等公民"的新观念所取代。Hadoop 集群在物理上和逻辑上融入企业的地方被称为"运行环境"。

影响 Hadoop 运行环境选择的因素有很多，已经超出了本书的范畴，我们将重点关注现在使用的典型操作环境。由于服务器和网络硬件的快速发展（感谢摩尔定律），Hadoop 可以在一些不同环境下运行。

内部（**In-house**）

　　该 Hadoop 环境包含企业自己拥有和运营的一系列物理机器，并且在企业自己掌控的数据中心里运行。

管理（**Managed**）

　　这种 Hadoop 环境是内部环境的一种演变，也由物理机器组成，但企业并不拥有和运营这些机器。它们是从另一个单独的企业租来的，该企业负责服务器的所有供给和维护工作。服务器在它们自己的数据中心里运行。

云（**Cloud**）

　　这种 Hadoop 环境与其他的相比完全不同，一个云环境由物理上分布在很多不同地点的虚拟服务器组成。最流行的 Hadoop 云服务是

亚马逊的 EC2（Elastic Compute Cloud，弹性计算云）。

3.2　网络安全

网络安全是一个详细的话题，此处无法彻底阐述。因此，我们将重点关注一些重要的、常用于保护 Hadoop 集群网络的网络安全主题，其中第一个就是网络划分。

3.2.1　网络划分

网络划分是将机器和服务从更大的网络中隔离出来的常见做法。不管我们讨论 Hadoop 集群、Web 服务器、部门桌面工作站还是其他系统，这种做法均适用。可以用两种不同的方法创建一个网段，这两种做法通常同时出现。

第一种方法是物理网络划分，使用路由器、交换机、防火墙等设备对网络的一部分进行分割。虽然这些设备运行在 OSI 模型的较高层，但从物理层角度看，网络分隔就是一个网段中的所有设备都物理接入独立于更大网络中其他设备的网络设备。

第二种方法是逻辑网络划分。逻辑划分运行在 OSI 模型的较高层，最普遍的是在使用 **IP** 寻址的网络层。在逻辑隔离下，同一网段的设备以某种方式组合到一起。实现这个目标的最普遍的方法是使用子网。例如，一个 Hadoop 集群有 150 个节点，有可能这些节点在逻辑上被分到同一个 /24 子网（即一个掩码为 255.255.255.0 的子网，包含最多 256 个 IP 地址，其中 254 个可用）。使用这种方法在逻辑上组织主机便于管理和安全保护。

网络划分方法中最常用的是，采用物理和逻辑网络划分的混合方法。实现这种混合方法的最常用方式是使用 **VLAN**（**virtual local area network**，虚拟局域网）。VLAN 允许多个网络子网共享物理交换机，虽然所有 VLAN 共享一个 2 层网络，但每个 VLAN 都是一个独立的广播域。根据网络交换机或路由器的能力，可能需要将每个物理端口分配到一个 VLAN，或者利用分组标记（packet tagging）在同一个端口运行多个 VLAN。

如前所述，物理隔离和逻辑隔离可以并且经常同时使用。物理和逻辑隔离可能存在于内部和管理环境，该环境中，Hadoop 集群具有一个给定的逻辑子网，并且所有机器都物理连接到同一组专用网络设备（例如 ToR 交换机和汇聚交换机）。

云运行环境中，物理上的网络划分通常要困难得多。云基础设施的设计目标就是要让硬件的位置变得不那么重要，更重要的是能够按需调整服务可用性。一些云环境允许用户选择处于同一位置组的机器，虽然从性能角度看这样当然更好（例如网络延迟更小），但通常无益于安全。同一位置组的机器可能与其他机器共享相同的物理网络。

现在，如果有一个 Hadoop 集群位于自己的网段，应当如何保护该网段呢？这很大程度上要靠网络防火墙和入侵检测与防护系统。

3.2.2　网络防火墙

网络防火墙将 Hadoop 集群从它所在的网络中强制隔离，是一种非常好的方法。防火墙的基本前提是，它被用作不同网段之间网络流量的附加安全层。例如，网络防火墙可能存在于公司内部用户网段和含有互联网站点的网段之间。同样，网络防火墙不太可能出现在一座办公楼同一个部门的两台桌面计算机之间。

表面上看，网络防火墙是不同于路由器、交换机等网络设备的另一种硬件，但也不完全如此。现在的路由器和（多层）交换机通常也具备许多与独立防火墙相同的核心功能。我们将在本节讨论 Hadoop 环境下关于防火墙的几个要点。

网络防火墙的基本功能是，根据网络和传输层属性允许或过滤（丢弃）网络数据包。归根结底就是，根据源和目的 IP、协议类型（如 TCP、UDP、ICMP）、源和目的端口（如果适用）进行过滤决策。网络设备很容易进行这些过滤决策，因为所有这些信息都包含在数据包头中，也就是说，并不必须对载荷数据进行深度包检测。

Hadoop 的基本过滤的价值通常见于 3 种常用类型：进出集群的数据传输、客户端访问（包括终端用户和第三方工具）、管理流量。每个类型都从不同视角考虑，如何用网络防火墙确保 Hadoop 集群和其他一切事物之间的网络路径的安全性。

01. 数据传输

    第一种类型是数据传输，即数据是如何被接收到集群或者提供到下游系统的。第 10 章将详细讨论从 Hadoop 生态系统的角度出发，如何保护这些流的安全。现在重点关注的是这些数据传输的网络通道和包含的数据类型，从而决定防火墙所需的检查级别。

首先看数据传输的网络通道，常见的选择是通用的文件传输工具，例如 FTP 和 SCP、 RDBMS 流（经由 Sqoop）、Flume 等流接收工具产生的数据流。这些常见的通道都有相应的 IP 地址和端口，可以很好地进行网络通信识别，以及创建允许该通信的防火墙规则。了解预期流量是什么样的很重要。哪些机器拥有源数据？数据在导入集群之前是否先要到集群边缘的一台服务器上？哪些机器在接收从集群提取的数据？解决这些问题要求网络防火墙规则大体上能够做到以下几点。

- 允许由特定 FTP 服务器向一个或多个边缘节点的 FTP 流量（本章将会进一步描述）；
- 允许集群中的工作节点通过指定端口连接一个或多个数据库服务器，以发送和接收数据；
- 允许数据通过限定数量的端口，从 Web 服务器集群产生的日志事件中传输到一系列 Flume 代理。

接下来要确定的是数据来源，以及是否需要额外的防火墙检测。例如，若一个上游数据源来自内部业务系统，那么上述防火墙策略就足够了。但若来自一个不可信的源（如互联网上提供的数据），则可能需要深度包检测，以保护集群不受恶意内容的危害。

05. 客户端访问

第二种常见的类型是关于客户端访问的。我们将在第 11 章详细讨论这个话题。但从网络防火墙的角度看，重要的是了解并区分客户端与集群交互所使用的方法。一些集群会在“全黑”环境中运行，这意味着不允许任何终端用户活动。这种类型的环境通常会运行持续的 ETL 作业，全自动产生结果数据并报告给下游系统。这种环境下，客户端访问策略只是简单地阻断一切，保持集群运行和安全只需要数据传输和管理相关的策略。

一种更加普遍的环境是，包含了访问集群的用户、工具和应用程序的混合环境。这种情况下，组织管理是关键。第三方工具运行在什么地方？它们能否被单独放到几个已知的机器？用户从什么地方访问集群？能否要求用户使用边缘节点？自定义应用程序运行在什么地方？网络防火墙处于应用程序和集群之间，还是处于应用程序和用户之间？

06. 管理流量

最后一个常见的种类是管理流量，包括管理员用户的登录行为、从集群到外部审计服务器的审计事件流量、从集群到另外一个网络的备份流量等。备份可能是通过 DistCp 进行大量的数据传输，甚至是将 Hive metastore 数据库备份到集群数据中心之外的地方。“管理流量”这个词并不是要描述量有多少，而是说该流量与常规客户端产生的流量有所不同。

网络防火墙是集群和外部网络之间的一种很好的安全边界，但如何防止可能主动攻击集群机器的恶意流量呢？对此，我们接下来讨论入侵检测和防御。

3.2.3　入侵检测和防御

上一节介绍了，如何使用网络防火墙作为控制进出 Hadoop 集群所在网络的数据流。虽然这种方法对“正常”的日常流量很有效，但如果发生不那么正常的事件，该怎么办呢？如果一个恶意的攻击者绕过了网络防火墙，并尝试攻击集群中的机器（如缓冲区溢出攻击），会怎样呢？如果是分布式拒绝服务攻击呢？入侵检测和防御系统可以帮助阻止这些类型的攻击。探讨入侵检测和防御设备如何融入集群的系统架构之前，先了解一些关于这些系统的基础知识。

讨论网络安全时，入侵检测系统（**IDS**）和入侵防御系统（**IPS**）经常被混为一谈。然而，这两个系统处理可疑入侵事件时，扮演的角色是根本不同的。IDS，顾名思义是检测入侵事件，与监控和预警系统属于同一类别。IDS 通常以混杂模式接入交换机，这意味着所有交换机上的网络流量在发送到指定目标端口的同时，也会发送给 IDS。当 IDS 发现一个被怀疑是攻击行为的数据包或者数据流时，就会产生一条警告。警告可能是发给独立的监控系统的事件，也可能只是一封安全管理员订阅的邮件。图 3-1 展示了一个包含 IDS 的网络图，可以看出，IDS 并不处于集群网络与外部网络之间的网络流中。

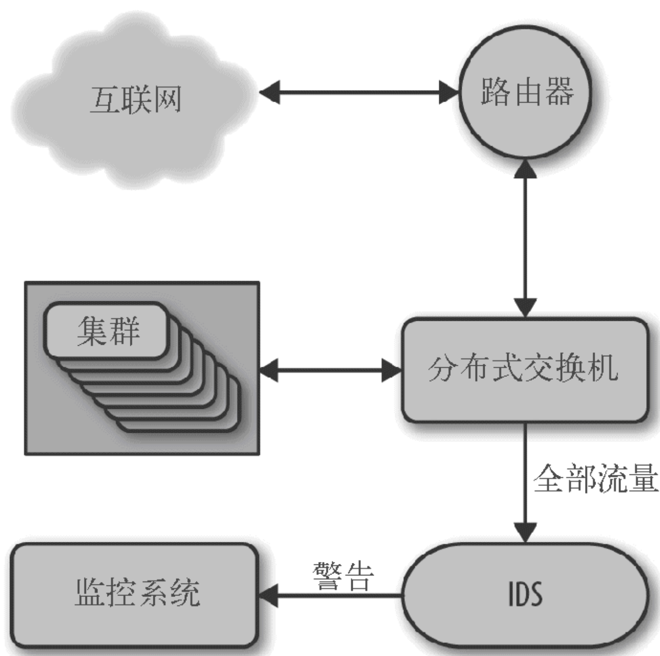


图 **3-1**：包含 **IDS** 的网络图

另一方面，IPS 不仅检测入侵行为，还会在入侵行为发生时主动尝试预防或阻止。之所以能做到这一点是因为，IDS 和 IPS 的关键区别在于，IPS 不在网络中以混杂模式监听数据，而处于网络两侧的中间。正因如此，IPS 可以阻断流向网络另一端的入侵数据流。IPS 的一个常见功能就是故障时关闭（**fail close**），这意味着，IPS 发生故障时（例如遭受大面积的 DDoS 攻击，导致其无法扫描数据包），它会直接阻断所有流向 IPS 另一侧的数据包。虽然这看上去可能也算一种成功的 DDoS 攻击，但在一定程度上，这种故障时关闭的机制反而保护了 IPS 后面的所有设备。图 3-2 展示了包含 IPS 的网络图，可以发现，IPS 实际上处于集群网络与外部网络之间的网络流中。

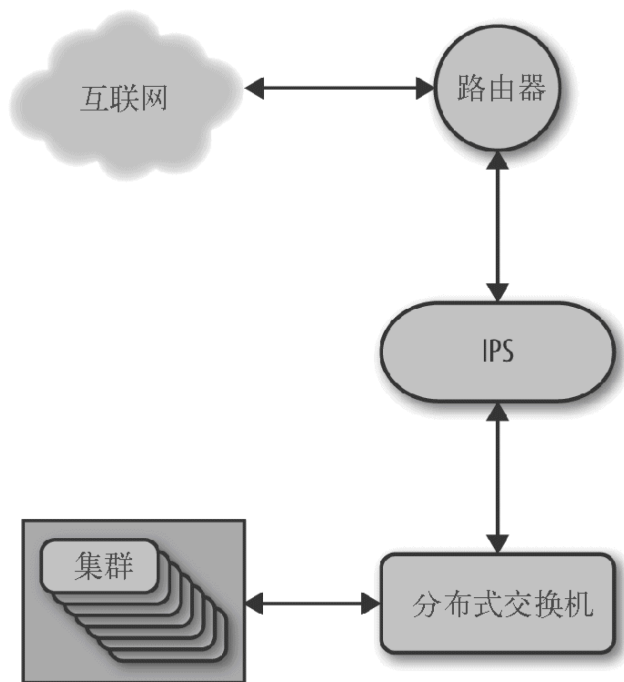


图 **3-2**：包含 **IPS** 的网络图

我们已经从宏观上了解了这些设备的作用，这对于 Hadoop 又有什么帮助呢？这其实是另外一个网络安全难题。Hadoop 集群本身存储着大量数据，对集群的入侵尝试进行检测和防御对保护大规模数据而言至关重要。那么，相对于 Hadoop 集群所在网络的其他部分而言，这些设备应当放在什么位置呢？答案是：可能是多个地方。

讨论网络防火墙的时候我们提到过，从安全角度看，来自开放互联网的数据接收通道需要被区别对待。对于 IDS 和 IPS 设备来说也同样如此，入侵攻击大部分来自互联网上的恶意用户。考虑到这一点，将 IPS 放到互联网数据源的接收路径中就十分合理了。互联网可能是恶意攻击者的温床，但企业的内部威胁也是实际存在的，且不容忽视。选择安全架构时，必须假定恶意攻击者也在同一座楼中工作。将 IDS 放在可信网络内部后，可以作为提醒管理员防范内部威胁的重要工具。

讨论 IDS 和 IPS 的一个额外的好处是，记录大量网络流也是很好的 Hadoop 用例。安全公司常常使用 Hadoop 收集很多不同用户网络中的 IDS 日志，以进行一系列的大规模数据分析和可视化，这也会反过

来扩充防火墙和 IDS/IPS 设备使用的规则引擎。

3.3 Hadoop角色和隔离策略

之前我们提到，集群中的节点可以被分成几组，以帮助建立充分的安全策略。本节看看如何做到这一点。每个节点在集群中都扮演着某种角色，这些角色决定了需要哪些安全策略来保护它。首先，回顾一下常见的 Hadoop 生态系统组件，以及每个组件承担的服务角色（我们假定你已经了解这些服务角色是做什么的，如果不是，请快速回顾第 1 章）。

HDFS

NameNode（活跃 / 待命 / 备用）、DataNode、JournalNode、FailoverController、HttpFS、NFSGateway

MapReduce

JobTracker（活跃 / 待命）、TaskTracker、FailoverController

YARN

ResourceManager（活跃 / 待命）、NodeManager、JobHistory Server

Hive

Hive Metastore Server、HiveServer2、WebHCatServer

Impala

Catalog Server、StateStore Server、Impalad

Hue

HueServer、Beeswax、KerberosTicketRenewer

Oozie

OozieServer

ZooKeeper

ZooKeeper Server

HBase

Master、RegionServer、ThriftServer、RESTServer

Accumulo

Master、TabletServer、Tracer、GarbageCollector

Solr

SolrServer

管理和监控服务

Cloudera Manager、Apache Ambari、Ganglia、Nagios、Puppet、Chef 等

通过这个（非详尽的）列表可以看到，许多生态系统项目都有主／从式架构，这很适合从安全架构角度组织服务角色。此外，还有一些服务角色是面向客户端的。总的来说，隔离策略是，确定运行在主节点上的主服务、工作节点上的工作服务和管理节点上的管理服务，还要确定哪些组件需要部署客户端配置文件，从而使得用户能够访问该服务。这些客户端配置文件与面向客户端的服务一起放到边界节点上。接下来将更详细地讲解节点的分类。

3.3.1　主节点

主节点可能是最重要的节点组，它包括所有作为 Hadoop 骨干的主要服务。由于这些角色对于它们所代表的组件的重要性，所以也需要更多安全策略保护自己。如下是需要在专用主节点中运行的角色列表：

- HDFS NameNode、备用 NameNode（或者待命 NameNode）、Failover Controller、JournalNode、KMS
- MapReduce JobTracker、FailoverController
- YARN ResourceManager、JobHistory Server
- Hive Metastore Server

- Impala Catalog Server、StateStore Server
- Sentry Server
- ZooKeeper Server
- HBase Master
- Accumulo Master、Tracer、GarbageCollector

有了这个服务列表后，第一个安全问题是：谁需要访问主节点，目的是什么？简单的答案是：管理员，为了执行管理功能。集群的客户——不管是实际的终端用户还是第三方工具——都可以利用开放的标准接口远程访问所有这些服务。例如，一个使用 `hdfs dfs -ls` 命令的用户可以在任何拥有相应 HDFS 服务客户端配置的机器上成功执行该命令，而不需要在运行 HDFS NameNode 的主节点上执行。考虑到这一点，需要将主节点限制为"仅限管理员访问"，原因如下。

资源连接

　　如果普通终端用户能够使用主节点运行任意程序，从而使用系统资源，那么这会夺走原本主节点角色所需的资源，从而导致性能下降。

安全漏洞

　　软件天生就具有漏洞，Hadoop 也不例外。如果允许用户访问担任主节点角色的机器，那就为利用 Hadoop 代码中未修补的漏洞（不管是恶意的还是无意的）打开了一扇大门。限制对主节点的访问降低了利用这些安全漏洞的风险。

拒绝服务

　　"用户可以做出不可思议的事情"，这已经是很客气的说法了。如果终端用户使用与主节点角色相同的机器，就不可避免地为其做出某些会搞垮主进程的行为创造条件。回到资源争用的问题。如果用户运行了一个占满日志目录的失控的进程，结果会怎么样？所有主节点角色都能处理无法记录日志的情况吗？管理员会想找出问题吗？另外一个类似的例子是，一个失控的进程占满了系统的 CPU 或者内存资源，从而导致系统出现内存不足的错误。

3.3.2　工作节点

工作节点负责处理 Hadoop 集群实际做的大部分工作，即存储和处理数据。工作节点上的典型角色如下。

- HDFS DataNode
- MapReduce TaskTracker
- YARN NodeManager
- Impala Daemon
- HBase RegionServer
- Accumulo TabletServer
- SolrServer

由于这些角色负责处理用户的数据和处理请求，所以表面上看，好像所有集群用户都需要访问这些节点，然而并非如此。通常只有管理员才需要远程访问工作节点以执行维护任务，而终端用户可以通过相关的接口和 API 进行数据接收、任务提交和记录检索。大多数情况下，服务会提供一种代理机制，允许管理员将用户活动定向到实际工作节点之外的一系列其他节点。具体细节稍后详细阐述。这些代理代表用户与工作节点进行通信，从而消除了直接访问的必要。

与主节点一样，合理的做法是仅限管理员访问工作节点，理由如下。

资源争用

　　当普通终端用户在工作节点上的预期进程之外进行活动时，可能造成资源管理失衡。例如，YARN 可被配置成使用特定数量的系统资源，具体数量是 Hadoop 管理员根据操作系统和其他软件的实际需要计算的。但终端用户的活动如何计算？通常很难精确描绘和计算用户活动，因此，很可能使用率高的工作节点相比未被使用的工作节点运行得不是很好，或者无法预测。

工作角色失衡

　　如果终端用户使用工作节点进行日常活动，可能造成工作节点角色行为上的不良失衡。例如，如果终端用户经常登录某一个特定的、运行着 DataNode 角色的工作节点，那么从该节点接收数据就会产生磁盘使用的失衡。因为 HDFS 写操作将会在选择集群中其他位置之

前，先尝试写入本地的第一个区块。这意味着，如果一个用户尝试上传一个 10 GB 的文件到 HDFS 的 home 目录，这 10 GB 都会被写到本地接收数据的 DataNode。

3.3.3 管理节点

管理节点是管理员的命脉。这些节点提供了安装、配置、监控和维护 Hadoop 集群的机制。这些节点上的典型角色如下。

- 配置管理
- 监控
- 警告
- 软件仓库
- 备份数据库

这些管理节点通常包含集群的软件仓库，尤其是在 Hadoop 集群节点没有互联网访问权限的情况下。管理节点最重要的职能是配置管理软件。不管是 Hadoop 专用的（如 Cloudera Manager、Apache Ambari）还是非专用的（如 Puppet、Chef），都是管理员建立和配置集群的地方。配置管理的必然结果是监控和警告，这些角色由 Ganglia、Nagios 以及 Hadoop 专用管理控制台等软件包提供。

此处仍要强调：这些节点不是给普通用户的。Hadoop 集群的管理和维护是一项管理职能，因此也应当被作为管理职能得到保护。尽管如此，也有一些例外。一种常见的例外是，开发者拥有集群监控仪表板的访问权限，从而在作业运行时监控相关指标，确认代码性能水平。

3.3.4 边界节点

边界节点受到所有 Hadoop 集群用户的关心，这些节点包含了最终为用户提供 Hadoop 存储和计算系统利用机制的 Web 接口、代理和客户端配置。边界节点上通常有如下角色。

- HDFS HttpFS 和 NFS 网关
- Hive HiveServer2 和 WebHCatServer
- Impala 网络代理 / 负载均衡器

- Hue Server 和 Kerberos 票据更新器
- Oozie Server
- HBase Thrift Server 和 REST Server
- Flume Agent
- 客户端配置文件

看到边界节点上常见的角色列表时，会明显发现，这种节点类型与其他节点有些不同。与其他节点类型的情况不同的是，各个边界节点通常都大相径庭。例如，使用 Flume 代理的接收管道通常会在用户不能访问的边界节点上，而包含便于命令行访问的客户端配置的边界节点则应当是用户可访问的。“边界节点组内的节点分类需要多么细致”取决于很多因素，包括集群规模和具体用例。以下是对边界节点进行进一步分类的例子。

数据网关

HDFS HttpFS 和 NFS 网关、HBase Thrift Server 和 REST Server、Flume 代理

SQL 网关

Hive HiveServer2 和 WebHCatServer、Impala 负载均衡代理（如 HAProxy）

用户门户

Hue Server 和 Kerberos 票据更新器、Oozie Server、客户端配置文件

> 虽然 Impala 守护进程不一定必须与 HDFS DataNode 搭配使用，但不建议使用独立的 Impala 守护进程作为代理。更好的选择是，使用负载均衡代理作为负载均衡器，如 HAProxy。当客户由于防火墙或其他限制无法直接连接工作节点上的 Impala 服务时，推荐使用这种架构。

使用上面给出的额外边界节点分类，可以很容易划分出哪些节点上用户理应有对其的远程访问权限，哪些节点只能通过配置的远程端口进

行远程访问。虽然用户需要远程访问门户节点，以通过控制台与集群进行交互，但数据和 SQL 网关不能通过这种方式访问是有道理的。这些节点只能通过远程端口访问，以便于访问在用户门户运行的命令行工具，和可能处在网络中其他位置的其他商业智能工具。

以上的分组只是示例，重要的是，不仅要了解安装在集群中的服务，还要知道这些服务是如何使用、被谁使用的。这又绕回之前关于了解用户和运行环境的讨论了。

3.4　操作系统安全

本节将探讨如何在操作系统层保护各个节点。

3.4.1　远程访问控制

典型的服务器环境中，远程访问控制比较简单。比如，一个运行 RDBMS 或 Web 服务的服务器通常是对终端用户锁定的，只允许特权用户以及管理员登录。而 Hadoop 环境没有这么简单。由于 Hadoop 生态系统的内在复杂性，除了基本管理的典型角色和职责之外，还有无数工具和访问方法可以与集群进行交互。虽然 Hadoop 集群可能包括成千上万的节点，但正如本节稍后将讲述的，这些节点是可以被分组的。考虑到这一点，确定哪些机器需要以及为什么需要被访问，并依此限制对机器的远程访问就很重要了。

3.4.2　主机防火墙

远程访问控制是限制哪些用户可以登录集群中特定机器的好方法。这固然是有用和必需的，但只是保护集群中特定机器方法的一小部分。主机防火墙是限制进出节点网络流量类型的一种十分有用的工具。Linux 系统中，主机防火墙通常是利用 iptables 实现的。当然也有其他第三方软件包可以实现该功能（如商业软件），但我们将关注的重点放在 **iptables** 上，因为它在大多数 Linux 发行版中都是默认提供的。

要使用 iptables，首先要对 Hadoop 集群中的网络流量进行理解和分类。表 3-1 展示了 Hadoop 生态系统组件常用的端口。我们将用这张表开始建立 iptables 的主机防火墙策略。

表3-1：Hadoop服务常见端口

组件	服务	端口
Accumulo	Master	9999
	GarbageCollector	50091
	Tracer	12234
	ProxyServer	42424
	TabletServer	9997
	Monitor	4560，50095
Cloudera Impala	Catalog Server	25020，26000
	StateStore	24000，25010
	Daemon	21000，21050，22000，23000，25000，28000
	Llama ApplicationMaster	15000，15001，15002
Flume	Agent	41414
HBase	Master	60000，60010
	REST Server	8085，20550
	Thrift Server	9090，9095
	RegionServer	60020，60030
HDFS	NameNode	8020，8022，50070，50470
	SecondaryNameNode	50090，50495
	DataNode	1004，1006，50010，50020，50075，50475
	JournalNode	8480，8485
	HttpFS	14000，14001
	NFS Gateway	111，2049，4242
	KMS	16000，16001
Hive	Hive Metastore Server	9083
	HiveServer2	10000
	WebHCat Server	50111
Hue	Server	8888
MapReduce	JobTracker	8021，8023，9290，50030
	FailoverController	8018
	TaskTracker	4867，50060
Oozie	Server	11000，11001，11443
Sentry	Server	8038，51000
Solr	Server	8983，8984
YARN	ResourceManager	8030，8031，8032，8033，8088，8090
	JobHistory Server	10020，19888，19890
	NodeManager	8040，8041，8042，8044
ZooKeeper	Server	2181，3181，4181，9010

有了以上常用端口后，我们应当了解需要实施多么严格的策略。配置 iptables 规则包括端口和 IP 地址，以及通信方向。一个最基本的防火墙策略允许任意主机访问被允许的端口，并且允许所有返回的（已建立连接的）流量。一个 HDFS NameNode 的 iptables 策略可能看起来和示例 3-1 差不多。

示例 **3-1**　基本的 NameNode iptables 策略

```
iptables -N hdfs
iptables -A hdfs -p tcp -s 0.0.0.0/0 --dport 8020 -j ACCEPT
iptables -A hdfs -p tcp -s 0.0.0.0/0 --dport 8022 -j ACCEPT
```

```
iptables -A hdfs -p tcp -s 0.0.0.0/0 --dport 50070 -j ACCEPT
iptables -A hdfs -p tcp -s 0.0.0.0/0 --dport 50470 -j ACCEPT
iptables -A INPUT -j hdfs
```

该策略很宽松，它允许所有主机（0.0.0.0/0）通过常用的 HDFS NameNode 服务端口连接到机器上。然而，这个策略可能太开放了。假定 Hadoop 集群节点都在子网 10.1.1.0/24 中，并且一个用于集群通信的专用边界节点设置在主机 10.1.1.254 上。此外，Web 控制台开启了 SSL。那么这个 NameNode 机器的调整后的 iptables 策略可能会如示例 3-2 所示。

示例 **3-2**　安全的 NameNode iptables 策略

```
iptables -N hdfs
iptables -A hdfs -p tcp -s 10.1.1.254/32 --dport 8020 -j ACCEPT
iptables -A hdfs -p tcp -s 10.1.1.254/32 --dport 8022 -j DROP
iptables -A hdfs -p tcp -s 10.1.1.0/24 --dport 8022 -j ACCEPT
iptables -A hdfs -p tcp -s 0.0.0.0/0 --dport 50470 -j ACCEPT
iptables -A INPUT -j hdfs
```

调整后的策略要严格得多。它允许任意用户通过 SSL（50470 端口）连接到 NameNode Web 控制台，仅允许集群机器通过专用的 DataNode RPC 端口（8022）连接到 NameNode，并且仅允许来自边界节点的流向 NameNode RPC 端口（8020）的用户流量。

> 为了使策略生效，可能需要将 iptables 跳转目标插入 `INPUT` 部分的特定行号。此处为了简便，仅展示了追加（Append）操作。

3.4.3　SELinux

另外一个经常被讨论的关于操作系统安全的内容是安全增强型 **Linux**（**SELinux**），最早由美国的情报机构——美国国家安全局（NSA）开发。SELinux 的前提是提供 Linux 内核增强，从而允许强制访问控制（**MAC**）的策略和实施。SELinux 大致可以配置为以下几种方式。

禁用（**Disabled**）

　　这种模式下，SELinux 不生效，也不给操作系统提供任何额外的安全级别。这是 Hadoop 中极为普遍的配置方式。

许可（**Permissive**）

这种模式下，SELinux 是启用状态，但不对系统进行保护，而在策略被违反时打印警告。对于从了解系统中工作负载类型入手，以建立自定义策略来说，这种模式很有用。

强制（**Enforcing**）

这种模式下，SELinux 是启用状态，并根据特定的 SElinux 策略对系统进行保护。

除了许可和强制两种启用模式之外，SELinux 还有两种不同的强制实施类型：针对性强制实施（targeted enforcement）和多级安全（**multilevel security**，**MLS**）。针对性强制实施仅针对特定进程，这意味着，这些进程具有相关策略进行保护，没有策略的进程则不会被 SELinux 保护。这种保护模式显然不那么严格。另一方面，MLS 则更为深入。总体而言，MLS 的前提是所有用户和进程都具有一个安全级别，同时，文件和其他对象都具有一个安全级别需求。MLS 仿照美国政府的分级标准，如绝密（Top Secret）、机密（Secret）、秘密（Confidential）和非密（Unclassified）。美国政府的分级系统中，这些级别形成一种等级架构。这种架构中，具有特定访问级别的用户具有访问较低级别信息的权限。例如，如果一个用户具有机密安全级，那么该用户被允许访问操作系统中机密、秘密和非密的对象——因为秘密和非密的安全级别都比机密低，但该用户无法访问标记为“绝密”安全级的对象。

这些听起来很好，但与 Hadoop 有什么关系呢？SELinux 能否被用作运行各种 Hadoop 生态系统组件的操作系统的额外保护层呢？简短的回答是：不太能。并不是说不可能，而是承认在这一点上缺乏 SELinux 安全集成以及相关（可被安全管理员部署到集群中的）策略创建方面的发展。Hadoop 生态系统的本质才是问题所在。目前有数以百计的组件、工具和其他部件以这样那样的方式集成到平台，并／或对平台进行增强。加进来的工具越多，想出一系列 SELinux 策略对其进行全部管理的难度就越大。

如果非要这么用，可能的方法是，将系统设置为许可模式，并在集群中运行相当于“普通”级别的、使用了尽可能多的给定环境中的典型工具的工作任务。这样运行一段合适的时间后，即可使用 SELinux 生成的警告开始建立策略。此处的问题是，这很快会变成一个枯燥乏味的过程，而且每当引入新的组件或功能时，都要对该过程进行修改。

3.5　小结

本章从定义 Hadoop 所在的操作系统环境开始，粗略地分析了 Hadoop 环境。然后从网络安全角度讨论了对该环境的保护，包括利用常见的安全实践（如网络划分）、介绍防火墙和 IDS/IPS 等网络安全设备。随后了解了如何根据运行的服务类型将 Hadoop 集群分成不同的节点组。最后给出了根据节点分组保护单个节点操作系统的建议。

第 4 章将介绍 Hadoop 安全架构中的一个基础组件：Kerberos。Kerbero 是企业系统中的一个关键成员，Hadoop 也不例外。下一章将会结束关于安全架构的讨论，为认证、授权和审计奠定基础。

第 4 章　Kerberos

第一次提 Kerberos 的时候，甚至那些有经验的系统管理员和开发者都会被吓到。依赖 Kerberos 的应用程序和系统经常会有很多需要解决相关问题的求助电话和故障单。本章将介绍基本的 Kerberos 概念，这对理解强认证是如何工作的很有必要；也会阐述 Kerberos 在第 5 章中的 Hadoop 认证中如何发挥重要作用。

那么 Kerberos 究竟是什么？在神话里，Kerberos 是 **Cerberus** 的希腊语，是一只守护地狱入口的三头巨犬，它确保没有人能在进入地狱后离开。从技术角度（这个角度更轻松一些）来说，Kerberos 是麻省理工学院开发的一个认证机制。Kerberos 发展成为大大小小的计算机系统强认证的实际标准，具有很多不同的实现，包括从 MIT 的 Kerberos 分发到微软的活动目录认证组件。

4.1　为什么是Kerberos

为什么 Hadoop 需要 Kerberos ？看看 Hadoop 认证的默认模型，原因就很明显了。提交给 Hadoop 一个用户名时，它会很高兴地相信你所说的一切，并且确保整个集群的所有其他机器也相信。

打个比方，如果在聚会中，一个人接近你并自我介绍为 Bill，你自然会相信他确实是 Bill。你怎么知道他确实是 Bill ？因为他是这么说的，而且你毫无疑问地相信了他。缺少 Kerberos 的 Hadoop 差不多也是这样，不同的是它还做了进一步类推，不仅相信他就是他自己所说的 Bill，还保证其他人也都相信。这是个问题。

Hadoop 的设计初衷是存储和处理 PB 级的数据。老话说得好，“能力越大，责任越大”。企业中的 Hadoop 不能再使用过于简单方法来识别（和信任）用户了。之前的比喻中，Bill 向你介绍了他自己。这时候，如果你要求看他的有效证件，并在收到证件之后（这很自然，因为每个人参加聚会时都会带证件……）利用数据库验证其有效性，会怎样呢？这就和 Hadoop 通过添加 Kerberos 认证引入的身份验证是一样的。

4.2　Kerberos概览

万事俱备，现在深入讨论和理解 Kerberos 是如何工作的。正如你可能想到的那样，Kerberos 的实现是一种客户端 / 服务端架构。对 Kerberos 组成部分进行详细分解之前，先介绍一些 Kerberos 术语。

首先，Kerberos 中的身份标识称为主体（**principal**），每个参与 Kerberos 认证协议的用户和服务都需要一个主体来唯一地标识自己。主体分为两种：用户主体和服务主体。用户主体名称（**user principal name**，**UPN**）代表常规用户，类似于操作系统中的用户名或者账号。服务主体名称（**service principal name**，**SPN**）代表用户需要访问的服务，例如特定服务器上的数据库。稍后会通过一个例子更清楚地说明 UPN 和 SPN 的关系。

接下来一个重要的 Kerberos 术语是域（**realm**）。一个 Kerberos 域就是一个身份验证管理域，所有主体都被分配到特定的 Kerberos 域。域确立了边界，这使得管理更为容易。

确定了什么是主体和域之后，下一步自然是弄清楚存储和控制这些信息的是什么。答案是密钥分发中心（**key distribution center**，**KDC**）。KDC 由三部分组成：Kerberos 数据库、认证服务（**authentication service**，**AS**）和票据授予服务（**ticket-granting service**，**TGS**）。 Kerberos 数据库存储的内容包含主体及其所属域的所有相关信息，数据库中的 Kerberos 域使用如下命名约定进行标识。

```
alice@EXAMPLE.COM
```

唯一标识了 Kerberos 域 `EXAMPLE.COM` 中用户（也叫作短名称）`alice` 的 UPN。依照惯例，域名总为大写。

```
bob/admin@EXAMPLE.COM
```

一种常规 UPN 的变形，标识了域 `EXAMPLE.COM` 中一个管理员 `bob`。UPN 中的斜杠（/）用于分隔短名称和管理员标识。通常会约定使用 `admin` 表示管理员，但稍后会看到，这也是可以配置的。

```
hdfs/node1.example.com@EXAMPLE.COM
```

该主体表示一个 `hdfs` 服务的 `SPN`，该服务在 Kerberos 域 `EXAMPLE.COM` 中的主机 `node1.example.com` 上。SPN 中的斜杠（/）用于分隔短名称 `hdfs` 和主机名 `node.example.com`。

 整个主体名都是大小写敏感的。例如 `hdfs/`

Node1.Hadoop.com@EXAMPLE.COM 和第三个例子中的主体是不一样的。一般情况下，最好在主体的域部分使用大写，其他部分使用小写。值得注意的是，SPN 中涉及的主机名当然也是小写的，在主机命名和 DNS（域名解析）时最好也这样。

KDC 的第二部分——AS——负责在客户端向 AS 发起请求时，向客户端发放票据授予票据。TGT 用于请求访问其他服务。

KDC 的第三部分——TGS——负责验证 TGT，并授予服务票据（**service tickets**）。服务票据允许认证过的主体使用应用服务器提供的服务，该服务通过 SPN 进行标识。下一节将介绍获得 TGT、提交给 TGS 以及获取服务票据的流程，现在只需了解 KDC 有两部分：AS 和 TGS，负责处理认证和访问服务的请求。

Kerberos 数据库中有一种特种格式的主体 krbtgt/<REALM>@<REALM>，例如 krbtgt/EXAMPLE.COM@EXAMPLE.COM。该主体是 AS 和 TGS 内部使用的，其关键之处在于，它被用来加密发放给客户端的 TGT 内容，这保证了 AS 发放的 TGT 只能被 TGS 验证。

表 4-1 简要提供了本章涉及的 Kerberos 术语及其缩写。

表4-1：Kerberos术语缩写

术语	缩写	描述
用户主体名称	UPN	定义 Kerberos 中一个用户的主体，格式为 <短名称>@<域> 或 <短名称>/admin@<域>
服务主体名称	SPN	定义 Kerberos 中特定主机上一个服务的主体，格式为 <短名称>/<主机名>@<域>
票据授予票据	TGT	认证后授予用户的一种特殊票据
密钥分发中心	KDC	Kerberos 服务器，包含 3 部分：Kerberos 数据库、AS 和 TGS
认证服务	AS	发放 TGT 的 KDC 服务
票据授予服务	TGS	验证 TGT 并授予服务票据的 KDC 服务

之前展示的是一些基本的 Kerberos 组件，这些组件是粗略理解认证机制所需的。Kerberos 本身是一个很深入和复杂的话题，值得用一整本书介绍。好在已经有人这么做了，如果你想更深入地了解 Kerberos，可以参考 Jason Garman 的著作 *Kerberos: The Definitive Guide*。

4.3 Kerberos工作流：一个简单示例

现在通过一个工作流示例，展示 Kerberos 大概是怎么工作的。首先定义所有出现的组件。

`EXAMPLE.COM`

Kerberos 域。

Alice

一个系统用户，其 UPN 为 `alice@EXAMPLE.COM`。

`myservice`

`server1.example.com` 上运行的一个服务，其 SPN 为 `myservice/server1.example.com@EXAMPLE.COM`。

`kdc.example.com`

Kerberos 域 `EXAMPLE.COM` 的 KDC。

Alice 要想使用 `myservice`，需要向 `myservice` 提供一个有效的服务票据，具体方法如下所述（为了简便，此处省略部分细节）。

(1) Alice 需要获取一个 TGT。为此，她向 `kdc.example.com` 上的 AS 发起一个请求，表明自己是主体 `alice@EXAMPLE.COM`。

(2) AS 做出响应，为主体 `alice@EXAMPLE.COM` 提供一个使用密钥（密码）加密的 TGT。

(3) 接收到加密信息后，Alice 被提示输入主体 `alice@EXAMPLE.COM` 的正确密码，从而解密信息。

(4) 成功解密包含 TGT 的信息后，Alice 向 `kdc.example.com` 上的 TGS 请求服务 `myservice/server1.example.com@EXAMPLE.COM` 的一个服务票据，并在请求中提供出 TGT 信息。

(5) TGS 对 TGT 进行验证，并提供给 Alice 一个使用主体 `myservice/server1.example.com@EXAMPLE.COM` 的密钥加密的服务票据。

(6) Alice 现在将服务票据提供给 `myservice`，`myservice` 随后使用 `myservice/server1.example.com@EXAMPLE.COM` 的密钥对其解密，并验证票据有效性。

(7) 由于 Alice 已经正确验证其身份，因此服务 `myservice` 允许其使用。

以上大致描述了 Kerberos 是如何工作的。显然，这是一个大大简化了的例子，很多底层的细节没有得到展示。图 4-1 展示了这个例子的序列图。

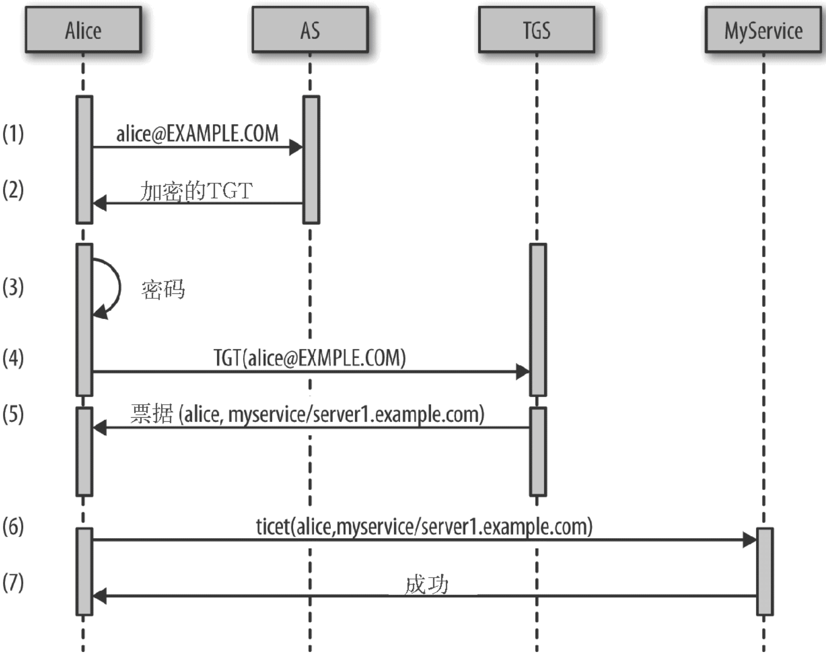


图 **4-1**：**Kerberos** 工作流示例

4.4　Kerberos信任

目前为止，我们介绍的 Kerberos 都默认所有用户和服务处于同一 Kerberos 域内。虽然这很适合入门，但考虑到庞大的公司规模，这通常是不符合实际的。随着时间的发展，大公司最终通过合并、收购，或仅仅想分隔公司的不同部分，将形成多个 Kerberos 域。然而，KDC 默认情况下只知道自身的域，以及自身数据库包含的主体。那么，当一个域中的用户想要使用另外一个域控制下的服务，该怎么办

呢？为了实现这个目标，两个域之间需要 Kerberos 信任（**trust**）。

例如，假定 Example 是一个很大的公司，并且决定创建多个域表示不同业务线，包括 HR.EXAMPLE.COM 和 MARKETING.EXAMPLE.COM。因为这两个域的用户有可能都需要访问两个域的服务，因此 HR.EXAMPLE.COM 的 KDC 需要信任来自 MARKETING.EXAMPLE.COM 域的信息，反之亦然。

表面上看起来很简单，但其实信任有两种类型：单向信任（**one-way trust**）和双向信任（**two-way trust**，或者 **bidirectional trust**、**full trust**）。我们刚才看的例子是一种双向信任。

如果还有一个 DEV.EXAMPLE.COM 域，其中的开发者主体需要访问 DEV.EXAMPLE.COM 和 MARKETING.EXAMPLE.COM 域，而营销用户不能访问 DEV.EXAMPLE.COM 域呢？这种场景就需要单向信任。单向信任在 Hadoop 部署中很常见，KDC 被安装配置成包含集群节点 SPN 的所有信息，但所有终端用户的 UPN 都在另外一个域，例如 Active Directory（活动目录）中。通常，Active Directory 管理员或公司政策会出于种种原因禁止双向信任。

那么 Kerberos 信任是如何实际建立的呢？本章之前提到过，有一种特殊的主体被 AS 和 TGS 内部使用，其格式是 krbtgt/< 域 >@< 域 >。这个主体在建立信任时更为重要，此时，该主体的格式变为 krbtgt/< 信任域 >@< 被信任域 >。该主体的关键之处在于，它在两个域中都存在。例如，如果 HR.EXAMPLE.COM 域需要信任 MARKETING.EXAMPLE.COM 域，那么主体 krbtgt/HR.EXAMPLE.COM@MARKETING.EXAMPLE.COM 需要在两个域中都存在。

> krbtgt/< 信任域 >@< 被信任域 > 主体的密码和加密类型必须与两个域中的都一样，以便建立信任。

上一个例子展示了单向信任中需要什么。为了建立双向信任，krbtgt/MARKETING.EXAMPLE.COM@HR.EXAMPLE.COM 主体也需要在两个域中都存在。总之，为了使 HR.EXAMPLE.COM 域和 MARKETING.EXAMPLE.COM 域具有双向信任，两个域都需要有 krbtgt/MARKETING.EXAMPLE.COM@HR.EXAMPLE.COM 和 krbtgt/HR.EXAMPLE.COM@MARKETING.EXAMPLE.COM 主体。

4.5 MIT Kerberos

正如本章开始提到的，Kerberos 由 MIT 首先创造。这些年来，它已历经数次修改，当前最新的版本是 MIT Kerberos V5，常称为 krb5。本节将介绍 MIT Kerberos 发行版的几个组件，用几个实际案例实践之前介绍的概念性例子。

要想获得关于 MIT Kerberos 发行版的最新权威资源，官方网站有很好的文档可以参考（http://web.mit.edu/~kerberos/）。

早先的例子里，我们忽略了一件事——Alice 发起了认证请求。实际上，Alice 是使用 **kinit** 工具做到这一点的。

示例 **4-1**　kinit（使用默认用户）

```
[alice@server1 ~]$ kinit
Enter password for alice@EXAMPLE.COM:
[alice@server1 ~]$
```

该例子将当前的 Linux 用户名 alice 和默认域合到一起，组成建议的主体 alice@EXAMPLE.COM。稍后深入讨论配置文件时，我们会讲解默认域。kinit 工具也允许用户显式地定义主体，以进行认证，如示例 4-2 所示。

示例 **4-2**　kinit（使用特定用户）

```
[alice@server1 ~]$ kinit alice/admin@EXAMPLE.COM
Enter password for alice/admin@EXAMPLE.COM:
[alice@server1 ~]$
```

如上所示，认证为管理员用户时，显式地提供主体名通常是必要的。另一个认证方法是使用 **keytab** 文件。keytab 文件存储了可被用于代替密码的加密密钥。创建 keytab 文件对于非交互式主体很有用，例如与长期运行着的进程（如 Hadoop 服务）相关的 SPN。keytab 文件并不需要一一映射到每个主体，多个不同主体的密钥可存储在一个 keytab 文件里。用户可以在使用 kinit 的时候指定 keytab 文件的路径，以及要认证为的主体名称（因为 keytab 文件中可能存在多个主体），如示例 4-3 所示。

示例 **4-3**　kinit（使用 keytab 文件）

```
[alice@server1 ~]$ kinit -kt alice.keytab alice/admin@EXAMPLE.COM
[alice@server1 ~]$
```

keytab 文件允许用户在没有密码相关知识的前提下进行认证。正因如此， keytab 需要用适当的控制手段加以保护，防止未授权用户使用其进行认证。keytab 是为管理员主体创建的时候，这尤为重要。

MIT Kerberos 发行版的另一个有用的功能称为 `klist`。该功能允许用户查看他们的证书缓存（**credentials cache**）中的 Kerberos 证书（如果有）。证书缓存存放在本地文件系统中，与成功通过 AS 验证的 TGT 的存储位置相同。默认情况下，该位置通常是 `/tmp/krb5cc_<uid>` 文件，其中 `<uid>` 是本地系统用户 ID 号。成功执行 `kinit` 后，alice 可以使用 `klist` 查看她的证书缓存，如示例 4-4 所示。

示例 **4-4**　使用 klist 查看证书缓存

```
[alice@server1 ~]$ kinit
Enter password for alice@EXAMPLE.COM:
[alice@server1 ~]$ klist
Ticket cache: FILE:/tmp/krb5cc_5000
Default principal: alice@EXAMPLE.COM

Valid starting     Expires            Service principal
02/13/14 12:00:27  02/14/14 12:00:27  krbtgt/EXAMPLE.COM@EXAMPLE.COM
        renew until 02/20/14 12:00:27
[alice@server1 ~]$
```

如果用户在没有进行认证的情况下尝试查看凭证缓存，则找不到任何凭证。

示例 **4-5**　找不到凭证

```
[alice@server1 ~]$ klist
No credentials cache found (ticket cache FILE:/tmp/krb5cc_5000
[alice@server1 ~]$
```

MIT Kerberos 工具箱中另一个有用的工具是 `kdestroy`。顾名思

义，它允许用户销毁凭证缓存中的凭证。这在切换用户、尝试或调试新配置的时候很有用（示例 4-6）。

示例 **4-6**　用 kdestroy 销毁凭证缓存

```
[alice@server1 ~]$ kinit
Enter password for alice@EXAMPLE.COM:
[alice@server1 ~]$ klist
Ticket cache: FILE:/tmp/krb5cc_5000
Default principal: alice@EXAMPLE.COM

Valid starting     Expires            Service principal
02/13/14 12:00:27  02/14/14 12:00:27  krbtgt/EXAMPLE.COM@EXAMPLE.COM
        renew until 02/20/14 12:00:27
[alice@server1 ~]$ kdestroy
[alice@server1 ~]$ klist
No credentials cache found (ticket cache FILE:/tmp/krb5cc_5000
[alice@server1 ~]$
```

到此为止，MIT Kerberos 的示例展示了其是“能用”的。隐藏在这些示例之外的是，为了使其真正工作起来，不管在客户端还是服务端都还需要很多必要的配置。接下来将介绍基本配置，从而把之前介绍的一些概念联系起来。

4.5.1　服务端配置

Kerberos 服务端配置主要在 `kdc.conf` 文件中，如示例 4-7 所示。该文件在 Red Hat/CentOS 系统中的位置是 `/var/kerberos/krb5kdc/`。

示例 **4-7**　kdc.conf

```
[kdcdefaults]
 kdc_ports = 88
 kdc_tcp_ports = 88

[realms]
 EXAMPLE.COM = {
  acl_file = /var/kerberos/krb5kdc/kadm5.acl
  dict_file = /usr/share/dict/words
  supported_enctypes = aes256-cts:normal aes128-cts:normal arcfour-hmac-mc
  max_renewable_life = 7d
}
```

配置文件的第一节 `kdcdefaults` 包含的配置适用于下面列出的所有域，除非某个域的配置包含同样配置项的值。配置项 `kdc_ports` 和 `kdc_tcp_ports` 分别指定了 KDC 监听的 UDP 和 TCP 端口。下一节 `realms` 则包含了使用该 KDC 作为服务器的所有域。一个 KDC 可以支持多个域。本例中，域的配置项包括如下内容。

`acl_file`

该项指定了管理服务用于进行访问控制的文件位置（稍后会详细介绍）。

`dict_file`

该项指定了包含不允许用作密码的单词的文件，这些密码是容易被破解 / 猜解的。

`supported_enctypes`

该项指定了 KDC 支持的所有加密类型。与 KDC 交互时，客户端必须支持该项列出的加密类型中的至少一种。谨慎使用 DES 等弱加密类型，因为这种加密很容易被攻破。

`max_renewable_life`

该项指定了票据可更新的最长时间。客户端可请求一个不超过该长度的更新有效期。典型的值是 7 天，用 7d 表示。

> 默认情况下，MIT Kerberos 中的加密选项通常被设成多个加密类型，其中包括 DES 等弱加密类型。可能的话，尽量删掉弱加密类型，以尽可能保证安全。弱加密类型容易被攻破，这已经被证明了。使用 AES-256 时，需要在集群的所有节点安装 Java Cryptographic Extensions，以允许不限强度加密类型。值得注意的是，一些国家禁止使用这些加密类型，请遵守你所在国家管理加密强度的法律。关于加密的更详细讨论参见第 9 章。

`acl_file` 所在位置（通常是 `kadm5.acl` 文件）用于控制哪些用户具有管理 Kerberos 数据库的权限。Kerberos 数据库管理员由两条不同但相关的组件控制：`kadmin.local` 和 `kadmin`。前者是允许 KDC 服务器的 `root` 用户修改 Kerberos 数据库的实用工具，顾名思义，它只能够由 Kerberos 数据库所在机器的 **root** 用户执行。想要远

程管理 Kerberos 数据库的管理员必须使用 kadmin 服务器。

Kadmin 服务器是一个允许远程连接并管理 Kerberos 数据库的守护进程，这也是 kadm5.acl 文件（示例 4-8）发挥作用的地方。Kadmin 工具使用 Kerberos 认证，而 kadm5.acl 文件则指定了哪些 UPN 被允许执行特权功能。

示例 **4-8**　kadm5.acl

```
*/admin@EXAMPLE.COM       *
cloudera-scm@EXAMPLE.COM *     hdfs/*@EXAMPLE.COM
cloudera-scm@EXAMPLE.COM *     mapred/*@EXAMPLE.COM
```

这允许来自 EXAMPLE.COM 的带有 /admin 标识的任意主体执行任意管理操作。虽然将 admin 标识改成其他任意名称也可以接受，但从简单性和可维护性起见，还是建议保持这种约定。管理员用户应当只使用自己的管理凭证进行具体的特许操作，就像在 Linux 中，管理员不能使用 **root** 用户的凭证进行日常非管理操作一样。

上例也说明，ACL 可以如何被定义以限制特定主体的权限。它展示了用户 clouderascm 可以执行任意操作，但仅限于在以 hdfs 和 mapred 开头的 SPN 上。这种语法类型有助于向第三方工具授予创建和管理 Hadoop 主体的权限，但不授予执行所有管理功能的权限。

如前所述，kadmin 工具允许 Kerberos 数据库的管理，该工具带给用户一个类似于控制台的接口，以输入各种命令和对 Kerberos 数据库进行操作（示例 4-9 ~ 示例 4-12）

示例 **4-9**　向 Kerberos 数据库添加一个新主体

```
kadmin: addprinc alice@EXAMPLE.COM
WARNING: no policy specified for alice@EXAMPLE.COM; defaulting to no polic
Enter password for principal "alice@EXAMPLE.COM":
Re-enter password for principal "alice@EXAMPLE.COM":
Principal "alice@EXAMPLE.COM" created.
kadmin:
```

示例 **4-10**　显示 Kerberos 数据库中一个主体的详细信息

```
kadmin: getprinc alice@EXAMPLE.COM
Principal: alice@EXAMPLE.COM
```

```
Expiration date: [never]
Last password change: Tue Feb 18 20:48:15 EST 2014
Password expiration date: [none]
Maximum ticket life: 1 day 00:00:00
Maximum renewable life: 7 days 00:00:00
Last modified: Tue Feb 18 20:48:15 EST 2014 (root/admin@EXAMPLE.COM)
Last successful authentication: [never]
Last failed authentication: [never]
Failed password attempts: 0
Number of keys: 2
Key: vno 1, aes256-cts-hmac-sha1-96, no salt
Key: vno 1, aes128-cts-hmac-sha1-96, no salt
MKey: vno1
Attributes:
Policy: [none]
kadmin:
```

示例 4-11　删除 Kerberos 数据库中的一个主体

```
kadmin: delprinc alice@EXAMPLE.COM
Are you sure you want to delete the principal "alice@EXAMPLE.COM"? (yes/no
Principal "alice@EXAMPLE.COM" deleted.
Make sure that you have removed this principal from all ACLs before reusin
kadmin:
```

示例 4-12　列出 Kerberos 数据库中所有主体

```
kadmin: listprincs
HTTP/server1.example.com@EXAMPLE.COM
K/M@EXAMPLE.COM
bob@EXAMPLE.COM
flume/server1.example.com@EXAMPLE.COM
hdfs/server1.example.com@EXAMPLE.COM
hdfs@EXAMPLE.COM
hive/server1.example.com@EXAMPLE.COM
hue/server1.example.com@EXAMPLE.COM
impala/server1.example.com@EXAMPLE.COM
kadmin/admin@EXAMPLE.COM
kadmin/server1.example.com@EXAMPLE.COM
kadmin/changepw@EXAMPLE.COM
krbtgt/EXAMPLE.COM@EXAMPLE.COM
mapred/server1.example.com@EXAMPLE.COM
oozie/server1.example.com@EXAMPLE.COM
yarn/server1.example.com@EXAMPLE.COM
zookeeper/server1.example.com@EXAMPLE.COM
kadmin:
```

4.5.2　客户端配置

默认的 Kerberos 客户端配置文件通常被命名为 `krb5.conf`，在 UNIX/Linux 系统的 `/etc/` 目录下。客户端应用（包括 `kinit` 工具）需要使用 Kerberos 时，随时读取该配置文件。示例 4-13 展示的 `krb5.conf` 配置文件是根据 Red Hat/CentOS 6.4 中自带配置文件执行的最小配置。

示例 **4-13**　krb5.conf

```
[logging]
 default = FILE:/var/log/krb5libs.log
 kdc = FILE:/var/log/krb5kdc.log
 admin_server = FILE:/var/log/kadmind.log

[libdefaults]
 default_realm = DEV.EXAMPLE.COM
 dns_lookup_realm = false
 dns_lookup_kdc = false
 ticket_lifetime = 24h
 renew_lifetime = 7d
 forwardable = true
 default_tkt_enctypes = aes256-cts aes128-cts
 default_tgs_enctypes = aes256-cts aes128-cts
 udp_preference_limit = 1

[realms]
 EXAMPLE.COM = {
 kdc = kdc.example.com
  admin_server = kdc.example.com
 }

 DEV.EXAMPLE.COM = {
  kdc = kdc.dev.example.com
  admin_server = kdc.dev.example.com
 }

[domain_realm]
 .example.com = EXAMPLE.COM
 example.com = EXAMPLE.COM
 .dev.example.com = DEV.EXAMPLE.COM
 dev.example.com = DEV.EXAMPLE.COM
```

该例中有几个不同的节。第一个是 `logging`，其含义不言而喻。它定义了会产生日志的各种 Kerberos 组件的日志文件存放位置。第二节是 `libdefaults`，包括通用的默认配置信息。本节更深入地看一下个性化配置。

`default_realm`

该字段定义了没有提供任何域的情况下应该使用什么 Kerberos 域，这与早先 `kinit` 例子中没有提供域的情况刚好是相符的。

`dns_lookup_realm`

可以用 DNS 指定使用哪个 Kerberos 域。

`dns_lookup_kdc`

可以用 DNS 寻找 KDC 的位置。

`ticket_lifetime`

该项指定了票据持续的时间，可以为 KDC 指定的最大值之内的任意时间长度。常用值是 24 小时，标为 `24h`。

`renew_lifetime`

该项指定了票据的可更新时间。票据可以在不进行客户端认证的情况下，由 KDC 进行更新。这必须在票据过期之前进行。

`forwardable`

该项指定了票据是否可以转发。这意味着，如果一个用户已经拥有一个 TGT，但登录到其他远程系统，那么 KDC 可以在无需重新认证的情况下向其重新分发一个 TGT。

`default_tkt_enctypes`

该项指定了向 AS 发送请求时加密会话密钥的方式。从左到右优先级依次降低。

`default_tgs_enctypes`

该项指定了向 TGS 发送请求时加密会话密钥的方式。从左到右优先级依次降低。

`udp_preference_limit`

该项指定了 UDP 包的最大值，超过该值后将切换为 TCP。要想强制使用 TCP，应将该值设为 1。

下一节是 realms，列出了客户端知道的所有 Kerberos 域。kdc 和 admin_server 配置项分别告诉客户端哪个服务器运行着 KDC 和 kadmin 进程。这些配置可以在主机名后指定端口，如果没有指定，就默认使用 88（KDC）和 749（admin_server）端口。该例展示了两个域。这是一种常用配置，此时，两个域之间存在单向信任，并且都需要被客户端知晓。该例中，可能 EXAMPLE.COM 域包含所有终端用户主体，DEV.EXAMPLE.COM 包含一个开发集群中的所有 Hadoop 服务主体。使用这种方式配置 Kerberos 时，能够使该开发集群的用户使用他们在 EXAMPLE.COM 域中已有的凭证，以访问 DEV.EXAMPLE.COM 域。

最后一节是 domain_realm，定义了 Kerberos 域与 DNS 名称的对应关系。第一项表示 example.com 域名下的所有主机都映射到 EXAMPLE.COM 域，第二项表示 example.com 自身映射到 EXAMPLE.COM 域。dev.example.com 和 DEV.EXAMPLE.COM 的情况与之类似。如果本节中没有找到匹配项，客户端就会尝试使用 DNS 名称的域名部分（转换为大写）作为域名。

4.6　小结

本章要点是，Kerberos 认证是一种多级的客户端 / 服务端过程，用以提供用户和服务的强认证。我们介绍了 MIT Kerberos 发行版，它是 Kerberos 的一种主流实现。即便本章描述了如何配置 MIT Kerberos 的一些细节，我们仍然强烈推荐你参考 MIT Kerberos 官方文档（http://web.mit.edu/~kerberos/），因为它是对最新版本的最新参考文档，也为安全管理员配置 Kerberos 环境提供了关于所有配置项的更详细的指南。

第 5 章将把目前为止涉及的 Kerberos 概念放入 Hadoop 以及 Hadoop 生态系统环境，进行更深的探索。

第二部分　验证、授权和审计

第 5 章　身份和验证

对于任何系统，保护数据的第一个必要步骤就是为每位用户提供一个唯一的身份，并且对用户声明的身份进行验证。验证和身份极其重要，因为如果身份验证体系不能够确信用户确实符合他们所声明的身份，则无法对数据进行访问控制。

本章将详细研究 Hadoop 服务如何管理身份验证和身份。首先研究身份的概念，以及 Hadoop 如何合并来自 Kerberos KDC、LDAP 和 Active Directory 域的信息，并提供一个关于分布式身份的综合视图。然后讲解 Hadoop 内部如何表征用户，以及从外部、全局的身份到内部用户角色的映射选项。接下来，回顾 Kerberos，并进一步详细研究 Hadoop 如何通过 Kerberos 实现强认证。之后，再继续研究一些核心组件如何使用基于用户名 / 密码组合的身份验证体系，以及分布式身份验证令牌在整个架构中是怎样的角色。最后，讨论用户模拟，并深入了解 Hadoop 身份验证机制的配置。

5.1　身份

Hadoop 生态系统环境中，身份是一个相对复杂的主题。这是由于，Hadoop 在尽可能地实现与权威身份源的松耦合。第 4 章介绍了 Kerberos 身份验证协议，接下来将对其进行重点介绍，因为 Kerberos 协议是 Hadoop 中默认使用的安全认证协议。虽然 Kerberos 为可靠的身份验证提供支持，但它对高级身份特征（如用户组或角色）几乎没有提供任何支持。特别是 Kerberos 只将身份表现为简单的、分为两部分的字符串（对于服务来说是分为三部分的字符串），这两部分分别为短名称和域。当赋予每位用户一个唯一的身份标识时，这种身份表现法虽然有用，但对于实现可靠的身份验证协议来说仍然不够。

除了用户身份之外，大多数计算系统还提供了组。组通常被定义为一批用户的集合。由于 Hadoop 的目标之一是集成已有的企业系统，所以它务实地使用可插拔系统提供传统的群组概念。

5.1.1　将Kerberos主体映射为用户名

深入了解 Hadoop 如何将用户映射到群组之前，需要讨论 Hadoop 如

何将 Kerberos 主体名称转换为用户名。回忆第 4 章中 Kerberos 使用一个两部分字符串（如 alice@EXAMPLE.COM）或三部分字符串（如 hdfs/namenode.example.com@EXAMPLE.COM），这类字符串包含了短名称、域，以及一个可选的实例名或主机名。为了简化用户名的使用，Hadoop 将 Kerberos 主体名映射至本地用户名。Hadoop 能够使用 krb5.conf 文件中的 auth_to_local 设置，也能够使用 core-site.xml 文件中的 hadoop.security.auth_to_local 参数对 Hadoop 专有规则进行配置。

hadoop.security.auth_to_local 的值被设为一个或多个从主体名到本地用户名的映射规则。规则可以是以 DEFAULT 或 RULE: 开头的字符串，其后跟着三部分内容：初始主体转换，接收过滤器、代换命令。特殊值 DEFAULT 只映射 Hadoop 本地域名称中的第一部分（如 alice/admin@EXAMPLE.COM 被 DEFAULT 规则映射为 alice）。

01. 初始主体转换

    初始主体转换由一个数值加代换字符串组成。该数值对应除域之外的主体成员数量。代换字符串则定义主体如何被初始地转换。变量 $0 将指代主体的域，$1 将指代第一个成员，$2 将指代第二个成员。关于初始主体转换的一些示例见表 5-1。初始主体转换的格式为 [< 数值 >:< 字符串 >]，输出称为“初始本地名称”。

    表**5-1**：初始主体转换示例

主体转换	**alice@EXAM PLE.com的**初始本地名称	**hdfs/ namenode.example.com@EXAMPLE.COM** 的初始本地名称
[1:$1.$0]	alice.EXAMPLE.COM	不匹配
[1: $1]	alice	不匹配
[2:$1 $2@$0]	不匹配	hdfs namenode.example.com@EXAMPLE
[2:$1@$0]	不匹配	hdfd@EXAMPLE.COM

02. 接收过滤器

    接收过滤器是个正则表达式。如果初始的本地名称（如规则的第一部分，即初始主体转换的输出）与正则表达式相匹配，那么代换命令则会在该字符串上执行。当且仅当整个字符串与正

则表达式匹配时，才会认为初始本地名称符合正则表达式。这相当于正则表达式以 ^ 开始，以 $ 结束。关于接收过滤器的一些示例见表 5-2。接收过滤器的格式为 (< 正则表达式 >)。

表**5-2**：接收过滤器示例

接收过滤器	alice.EXAMPLE.COM	hdfs@EXAMPLE.COM
(.*\.EXAMPLE\.COM)	匹配	不匹配
(.*@EXAMPLE\.COM)	不匹配	匹配
(.*EXAMPLE\.COM)	匹配	匹配
(EXAMPLE\.COM)	不匹配	不匹配

03. 代换命令

代换命令是一个类似 sed 的批量替换命令，该命令由正则表达式和替换字符串组成。可以将一部分正则表达式用括号括起来表示匹配组，并在替换字符串中使用数字（例如 \1）对其进行引用。组号是由左括号在正则表达式中的顺序决定的。表 5-3 是代换命令的一些示例。代换命令的格式为 s/< 模式 >/< 替换字符串 >/g。其中，末位的 g 是可选的，若 g 存在，则表示对于整个字符串来说是全局代换；若 g 不存在，则表示只有匹配规则的第一个子串被代换。

表**5-3**：代换命令示例

代换命令	alice.EXAMPLE.COM	hdfs@EXAMPLE.COM
s/(.*)\.EXAMPLE.COM/\1/	alice	不可用
s/(.EXAMPLE.COM//	alice	hdfs
s/E/Q/	alice.QXAMPLE.COM	hdfs@QXAMPLE.COM
s/E/Q/g	alice.QXAMPLQ.COM	hdfs@QXAMPLQ.COM

一条规则的完整格式为：RULE:[< 数字 >:< 字符串 >](< 正则表达式 >)s/< 模式 >/< 替换字符串 >/。多条规则会分行隔开，且会按顺序执行。一旦某一主体完全匹配某一规则（如主体符合初始主体转换的数值，且初始本地名称符合接收过滤器），该规则就会输出该主体的用户名，并且不会继续执行其他规则。由于这样的顺序限制，我们通常将 DEFAULT 规则列在最后。

auth_to_local 设置最常见的使用方式是，用于配置如何处

理来自其他 Kerberos 域的主体。一个常见的使用场景是，设置一个或多个信任域。例如，用户的 Hadoop 域为 HADOOP.EXAMPLE.COM，但合作域为 CORP.EXAMPLE.COM，那么用户应添加能将合作域主体转换为本地用户的规则。示例 5-1 给出了仅接受来自 HADOOP.EXAMPLE.COM 域和 CORP.EXAMPLE.COM 域的用户，并且将这两个域的用户映射为两个域首部的配置样例。

示例 **5-1**　使用 auth_to_local 配置信任域

```
<property>
  <name>hadoop.security.auth_to_local</name>
  <value>
    RULE:[1:$1@$0](.*@CORP.EXAMPLE.COM)s/@CORP.EXAMPLE.COM//
    RULE:[2:$1@$0](.*@CORP.EXAMPLE.COM)s/@CORP.EXAMPLE.COM//
    DEFAULT
  </value>
</property>
```

5.1.2　Hadoop用户到组的映射

Hadoop 使用一个名为 hadoop.security.group.mapping 的配置参数，以控制用户到用户组的映射。默认的实现方法使用标准 UNIX 接口进行本地调用，或者使用 shell 命令查找用户到组的映射。这意味着，只有配置在调用映射的那台服务器上的用户组才对 Hadoop 可见。在实际操作中，这不是个大问题，因为 Hadoop 集群对访问集群的用户和用户组会有一致的视图。

> 除了要了解用户到组的映射机制如何工作之外，了解这些映射的发生位置也同样重要。正如第 6 章将讲到的，用户到组的映射要始终保持一致，并且要在进行授权决策的环节发生。对于 Hadoop，这意味着映射发生在 NameNode、JobTracker（对 MR1）以及 ResourceManager（对 YARN/MR2）进程中。这是个很重要的细节，因为默认的用户到组的映射实现是通过使用标准 UNIX 接口决定组成员的；对于存在于 Hadoop 视角中的用户组，也必须存在于运行有 NameNode、JobTracker 和 ResourceManager 的服务器上。

hadoop.security.group.mapping 的配置参数能够设置于任意

实现了 org.apache.hadoop.security.GroupMappingServiceProvider 接口的 Java 类中。除了之前介绍过的默认配置， Hadoop 还提供了很多对于该接口的实现，总结如下。

JniBasedUnixGroupsMapping

一个基于 JNI（Java Native Interface，Java 本地接口）的实现，调用 libc 库函数 getpwnam_r() 和 getgrouplist() 确定组成员。

JniBasedUnixGroupsNetgroupMapping

JniBasedUnixGroupsMapping 的一种扩展，通过调用库函数 setnetgrent()、getnetgrent() 和 endnetgrent() 确定网络组成员。只有服务分层授权访问控制列表中使用到的网络组才会被引入这个映射。

ShellBasedUnixGroupsMapping

一种基于 shell 的实现，使用 id -Gn 命令。

ShellBasedUnixGroupsNetgroupMapping

ShellBasedUnixGroupsMapping 的一种扩展，使用 getent netgroup shell 命令确定网络组成员。只有服务分层授权访问控制列表中使用到的网络组才会被引入这个映射。

JniBasedUnixGroupsMappingWithFallback

JniBasedUnixGroupsMapping 类的封装，若不能加载本地库（这是默认设置下的实现），则会回退到 ShellBasedUnixGroupsMapping 类。

JniBasedUnixGroupsNetgroupMappingWithFallback

JniBasedUnixGroupsNetgroupMapping 类的封装，若不能加载本地库，则会回退到 ShellBasedUnixGroupsNetgroupMapping 类。

LdapGroupsMapping

与 LDAP 或 Active Directory 服务器直连，确定组成员。

>  无论组映射是如何配置的，Hadoop 都会将映射内容缓存，并且仅在缓存记录过期后才调用映射的实现。组映射缓存默认为每隔 300 秒（5 分钟）过期。若用户想使底层组的更新更频繁地在 Hadoop 中体现，则要将 `core-site.xml` 中的 `hadoop.security.groups.cache.secs` 设置为希望记录被缓存的秒数。这个数值应设得尽量小，以迅速反映更新的变化。然而又不能太小，否则将导致对 LDAP 服务器或其他用户组提供者的不必要的访问。

使用LDAP将用户映射至组

大多数应用部署可以使用默认的组映射来源。然而，对于仅能直接从 LDAP 或 Active Directory 服务器——而不是从集群节点——引用群组的环境，Hadoop 提供了 `LdapGroupsMapping` 实现。该方法的配置通过设置位于 NameNode、JobTracker 和 / 或 ResourceManager 上的 `core-site.xml` 文件中的数个参数实现。

`hadoop.security.group.mapping.ldap.url`

　　用于解析用户群组的 LDAP 服务器 URL。必须以 `ldap://` 或者 `ldaps://`（若 SSL 功能开启）开头。

`hadoop.security.group.mapping.ldap.bind.user`

　　连接 LDAP 服务器时给用户绑定的唯一名称。该用户需要对目录进行读访问，并且不能是管理员。

`hadoop.security.group.mapping.ldap.bind.password`

　　已绑定用户的密码。最好不要使用该参数设置密码，而将密码放进一个独立的文件，并且将 `hadoop.security.group.mapping.ldap.bind.password.file` 指向该独立文件的路径。

> 
>
> 如果将 Hadoop 配置为直接使用 LDAP，则会缺失 Hadoop 服务账户（如 hdfs）的本地组。这样会导致

log 日志中产生大量如下消息：

```
No groups available for user hdfs
```

因此，使用 JNI 或基于 shell 的映射，并在操作系统层将 LDAP/Active Directory 集成的做法通常更好。系统安全服务守护进程（SSDD）融合了许多身份和验证系统，并且能为缓存和离线访问提供通用支持。

使用上述参数，示例 5-2 展示了如何在 `coresite.xml` 中实现 `Ldap Groups Mapping`。

示例 **5-2**　core-site.xml 中的 LDAP 映射示例

```
...
<property>
  <name>hadoop.security.group.mapping</name>
  <value>org.apache.hadoop.security.LdapGroupsMapping</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.url</name>
  <value>ldap://ad.example.com</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.bind.user</name>
  <value>Hadoop@ad.example.com</value>
</property>
<property>
  <name>hadoop.security.group.mapping.ldap.bind.password</name>
  <value>password</value>
</property>
...
```

除了必需的参数外，还有几个可选参数能够控制用户和组之间是如何映射的。

`hadoop.security.group.mapping.ldap.bind.password.file`

本参数为含有绑定用户密码的文件路径。该文件应该只能被运行服务（通常是 `hdfs`、 `mapred` 和 `yarn`）的 UNIX 用户读取。

`hadoop.security.group.mapping.ldap.ssl`

本参数值为 true 时，开启连接 LDAP 服务器的 SSL 协议。若该

设定为生效状态，则 hadoop.security.group.mapping.ldap.url 参数必须以 ldaps:// 起始。

hadoop.security.group.mapping.ldap.ssl.keystore

本参数为 Java 密钥库的路径，包含开启 SSL 时 LDAP 服务器要求的客户证书。该密钥库必须为 JKS（Java 密钥库）格式。

hadoop.security.group.mapping.ldap.ssl.keystore.password

本参数为 hadoop.security.group.mapping.ldap.ssl.keystore 文件的密码。最好不要使用该设置，而将密码放在另一个独立的文件中，并将 hadoop.security.group.mapping.ldap.ssl.keystore.password 属性配置为指向该独立文件的路径。

hadoop.security.group.mapping.ldap.ssl.keystore.password

本参数为包含 hadoop.security.group.mapping.ldap.ssl.keystore 文件密码的文件路径。该文件应该只能被运行服务（通常是 hdfs、mapred 和 yarn）的 UNIX 用户读取。

hadoop.security.group.mapping.ldap.base

该参数为用于搜索 LDAP 目录的搜索库。搜索库的名称唯一，且通常会在能够覆盖访问集群的所有用户的前提下被配置得尽可能具体。

hadoop.security.group.mapping.ldap.search.filter.user

用于搜索 LDAP 用户的过滤器。默认配置是 (&(objectClass=user)(sAMAccountName={0}))，通常适用于安装了 Active Directory 的情况。对于其他 LDAP 服务器，需要修改该配置。对于 OpenLDAP 和相应的兼容服务器，推荐的配置为 (&(objectClass=inetOrgPerson)(uid={0}))。

hadoop.security.group.mapping.ldap.search.filter.group

用于搜索 LDAP 用户组的过滤器。默认配置为

(objectClass=group)，通常适用于安装了 Active Directory 的情况。

hadoop.security.group.mapping.ldap.search.attr.member

组对象用于标识组内用户身份的属性。

hadoop.security.group.mapping.ldap.search.attr.group.nam

组对象用于标识组名的属性。

hadoop.security.group.mapping.ldap.directory.search.time

等待搜索目录结果的超时时间，以毫秒为单位。

5.1.3 Hadoop用户配置

最难理解的 Hadoop 安全需求之一就是，集群的所有用户都必须能在集群中的所有服务器上进行配置。这意味着他们既可以存在于本地的 /etc/passwd 文件中，也可以通过服务器接入的网络目录服务进行配置（这种情况更普遍），例如 OpenLDAP 或者 Active Directory。为了理解该需求，需要牢记，Hadoop 实际上是一个允许用户在集群机器上提交和执行任意代码的服务。这意味着，如果不信任用户，则需要限制这些用户对任何或所有运行在集群服务器上的服务进行访问，包括本地文件系统等标准 Linux 服务。目前，加强这些限制的最好方法是，在集群上使用提交任务的用户的用户名和 UID 执行各项任务。为了满足此需求，必须让集群中的每一台服务器都使用一致的用户数据库。

> 虽然所有集群用户都必须在集群上的所有服务器上进行配置，但不一定要给所有用户都开启本地或远程 shell 访问。最好的策略是，通过预设的 `shell/sbin/nologin` 对用户进行配置，并且通过 `/etc/ssh/sshd_config` 文件中的 `AllowUsers`、`DenyUsers`、`AllowGroups` 和 `DenyGroups` 配置禁止用户的 SSH 访问。

5.2 身份验证

Hadoop 的早期版本以及相关生态系统的项目是不支持强认证的。Hadoop 是个复杂的分布式系统，幸运的是，它的生态系统中，绝大

多数组件都根据有限的几个认证方式进行了标准化，这些标准化取决于具体的服务和协议。特别地，Kerberos 被用于生态系统中的绝大部分组件内，因为 Hadoop 在早期的安全属性开发时就已经根据它进行了标准化。表 5-4 是根据服务和协议总结的身份验证方法。本节重点关注 HDFS、MapReduce、YARN、HBase、Accumulo 和 ZooKeeper 的身份验证。Hive、Impala、Hue、Oozie 和 Solr 的身份验证将延后至第 11 章和第 12 章进行介绍，因为这些通常是由用户直接访问的。

表5-4：Hadoop生态系统的身份验证方法

服务	协议	方法
HDFS	RPC	Kerberos，委托令牌
HDFS	Web UI	SPNEGO（Kerberos），可插拔式
HDFS	REST（WebHDFS）	SPNEGO（Kerberos），委托令牌
HDFS	REST（HttpFS）	SPNEGO（Kerberos），委托令牌
MapReduce	RPC	Kerberos，委托令牌
MapReduce	Web UI	SPNEGO（Kerberos），可插拔式
YARN	RPC	Kerberos，委托令牌
YARN	Web UI	SPNEGO（Kerberos），可插拔式
Hive Server 2	Thrift	Kerberos，LDAP（用户名 / 密码）
Hive Metastore	Thrift	Kerberos，LDAP（用户名 / 密码）
Impala	Thrift	Kerberos，LDAP（用户名 / 密码）
HBase	RPC	Kerberos，委托令牌
HBase	Thrift Proxy	无
HBase	REST Proxy	SPNEGO（Kerberos）
Accumulo	RPC	用户名 / 密码，可插拔式
Accumulo	Thrift Proxy	用户名 / 密码，可插拔式
Solr	HTTP	基于 HTTP 容器
Oozie	REST	SPNEGO（Kerberos，委托令牌）
Hue	Web UI	用户名 / 密码（数据库、PAM、LDAP）、SAML、OAuth、SPNEGO（Kerberos）、远程用户（HTTP 代理）
ZooKeeper	RPC	用户名 / 密码）、IP、SASL（Kerberos）、可插拔式

5.2.1　Kerberos

Hadoop 支持两种身份验证机制：简单机制和 `Kerberos` 机制。简单机制是默认设置，根据客户进程的有效 UID 确定用户名，该用户名会不附带任何额外凭证，直接传给 Hadoop。这种模式下，Hadoop 服务完全信任他们的客户。这种默认设置允许任何可访问集群的用户均能被完全信任，直接访问上述集群的所有数据和管理员级功能。对于概念验证性系统或实验环境，通常会允许系统以这种模式运行，并使用防火墙对其进行保护，同时会限制那些能够使用客户端访问集群内任何系统的用户集。然而对于产品级或者多租户的系统，这样的身

份验证机制几乎不能被接受。简单认证同样被 HBase 作为其默认的身份验证机制。

HDFS、MapReduce、YARN、HBase、Oozie 和 ZooKeeper 都支持 Kerberos 作为客户端的身份验证机制，虽然在实现方面会由于服务和接口的区别而有些不同。对基于 RPC 的协议，简单认证与安全层（**Simple Authentication and Security Layer**，**SASL**）框架被用于向下层协议添加认证。理论上，任意 SASL 协议都能得到支持；但实际上，真正仅支持的机制是 GSSAPI（Kerberos V5）和 DIGEST-MD5（DIGEST-MD5 的详细内容参见 5.2.3 节）。Oozie 没有 RPC 协议，取而代之的是，它为用户提供了 REST 接口。Oozie 使用简单和受保护的 **GSSAPI** 协商机制（**Simple and Protected GSSAPI Negotiation Mechanism**，**SPNEGO**），该协议最初由微软在 Internet Explorer 5.0.1 和 IIS 5.0 中为 HTTP 层的 Kerberos 认证实现。HDFS、MapReduce、YARN、Oozie 和 Hue 的 Web 接口，以及 HDFS（包括 WebHDFS 和 HttpFS）和 HBase 的 REST 接口也同样支持 SPNEGO。SASL 和 SPNEGO 的身份验证过程遵循标准的 Kerberos 协议，只有提交服务票据的机制发生改变。

让我们来看 Alice 是如何通过 Kerberos 与 HDFS 的 NameNode 进行验证交互的：

- Alice 为由 `hdfs/namenode.example.com@EXAMPLE.COM` 标识的 HDFS 服务，从 `kdc.example.com` 的 TGS 请求一个服务票据，并随请求附上她的 TGT。

- TGS 验证 TGT 之后，提供给 Alice 一个服务票据，使用 `hdfs/namenode.example.com@EXAMPLE.COM` 的主体密钥。

- Alice 把服务票据提交给 NameNode（通过 SASL），NameNode 能够使用 `hdfs/namenode.example.com@EXAMPLE.COM` 的密钥对票据进行解密和认证。

5.2.2 用户名和密码验证

ZooKeeper 支持基于用户名和密码的认证。相比于使用一个用户名和密码的数据库，ZooKeeper 将验证密码的环节推迟到授权步骤（详见 6.4 节 ZooKeeper ACLs）。一个 ACL（Access Control List，访问控制列表）被附加到一个 ZNode 上时，会引入验证方案和方案 ID。这

个 ID 是该方案的认证提供者验证过的。用户名和密码的认证通过摘要式身份验证提供者实现，它会生成用户名和密码的一个 SHA-1 摘要值。由于认证环节被推迟到授权步骤，因此认证步骤总能通过。用户通过调用 `addAuthInfo(String scheme, byte[] authData)` 方法添加他们的验证细节，其中 `<username>:<password>.getBytes()` 作为鉴别数据时，`<username>` 和 `<password>` 会被相应的值替换。

Accumulo 也支持基于用户名和密码的验证。与 ZooKeeper 不同，Accumulo 使用更常见的方法存储用户名和密码，并且有一个清晰明确的登录步骤验证密码有效性。Accumulo 的验证体系对 `AuthenticationToken` 接口进行不同的实现，以保证其可插拔性。最常见的一种实现是 `PasswordToken` 类，该类可以从 `CharSequence` 或 Java 属性文件初始化。使用用户名和密码连接 Accumulo 的样例代码如示例 5-3 所示。

示例 **5-3**　使用用户名和密码连接 Accumulo

```
//创建一个Accumulo实例的句柄
Instance instance = new ZooKeeperInstance("instance",
    "zk1.example.com,zk2.example.com,zk3.example.com");

// 创建一个包含密码的令牌
AuthenticationToken token = new PasswordToken("secret");

//创建连接器；如果密码无效，则会抛出AccumuloSecurityException异常
Connector connector = instance.getConnector("alice", token);
```

5.2.3　令牌

任何分布式系统内，以用户名义执行的所有行为都有必要验证该用户身份。仅仅验证服务的归属者是不够的，验证必须发生在每一次交互中。以运行一个 MapReduce 作业为例，想要扩展命令行参数中的任意通配符，需要在客户端和 NameNode 之间进行身份验证；为了提交作业，则需要在客户端和 JobTracker 之间均进行身份验证。

JobTracker 将工作分解为被每个集群中的 TaskTracker 顺序执行的任务，每个任务必须与 NameNode 通信，以打开组成输入分片（input split）的文件。为了加强文件系统的许可控制，每个任务必须通过 NameNode 的身份验证。若 Kerberos 是唯一的身份验证机制，用户的 TGT 需要分派给每一个任务。该方法的缺点在于，这样做会允许任务向任意一个受 Kerberos 保护的服务发起身份验证请求，而这种

做法是不合适的。通过签发能分派给每个任务、但被限制在某个特定服务内的认证令牌，Hadoop 解决了该问题。

01. 委托令牌

Hadoop 有多种令牌，用于在没有 TGT 或 Kerberos 服务票据的情况下进行认证后的访问。通过 Kerberos 与 NameNode 完成身份验证后，一个客户端能够获得一个委托令牌。委托令牌其实是客户端和 NameNode 之间共享的密钥，能够通过 DIGEST-MD5 机制用于 RPC 身份验证。

图 5-1 展示了客户端和 NameNode 之间的两次交互。首先，客户端使用 Kerberos 服务票据进行了 `getDelegationToken()` 的 RPC 调用，以请求一个委托令牌 (1)。NameNode 回复了一个委托令牌 (2)。客户端发起 `getListing()` 的 RPC 调用，以请求目录列表，但这次客户端使用的是供身份验证用的委托令牌。验证令牌后，NameNode 会给客户端响应，内容为 `DirectoryListing`(4)。

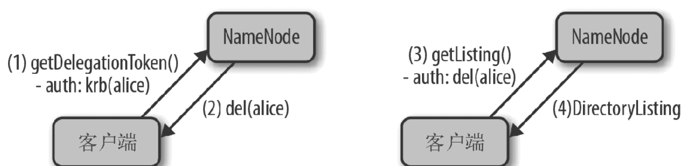


图 **5-1**：获取并使用委托令牌

令牌既有失效日期，也有最大发行日期。令牌在失效日期之后会失效，但直至其最大发行日期前，仍然能够被更新。客户端能够在完成向 NameNode 的初始 Kerberos 认证之后，发起对委托令牌的请求。令牌会有一个指定的更新机制。为用户更新令牌时，令牌更新器使用其 Kerberos 凭证进行认证。委托令牌最常见的用法是在 MapReduce 任务中，客户端指定 JobTracker 为令牌更新器。委托令牌由 NameNode 的 URL 生成，且存储在 JobTracker 的系统目录下，以便将其传给任务。该方法使得任务访问 HDFS 时，不会将用户的 TGT 置于泄露风险中。

02. 块访问令牌

文件权限检查由 NameNode 执行，而非 DataNode。默认情况下，任何客户端只要提供块 ID 就可以访问任意块。为了解决这个问题，Hadoop 引入了块访问令牌的概念。块访问令牌由 NameNode 生成，在客户端通过身份验证，且 NameNode 已执行完访问文件 / 块的必要授权检查后，会将其发送给客户端。块访问令牌包括客户端 ID、块 ID 和允许访问模式（READ、WRITE、COPY、REPLACE），并且使用 NameNode 和 DataNode 之间的共享密钥进行签名。该共享密钥永远不会让客户端获得，且块访问令牌过期时，客户端必须向 NameNode 重新请求一个令牌。

图 5-2 展示了客户端如何使用块访问令牌读取文件。首先，客户端使用 Kerberos 凭证，通过 `getBlockLocations()` 的 RPC 调用，向 NameNode 请求所需的块位置 (1)。NameNode 会响应一个 `LocatedBlock` 对象，该对象包括针对被请求块的块访问令牌 (2)。客户端使用块访问令牌，通过数据传输协议中的 `readBlock()` 方法，向 DataNode 进行身份验证并请求数据 (3)。最后，DataNode 将响应请求的数据 (4)。

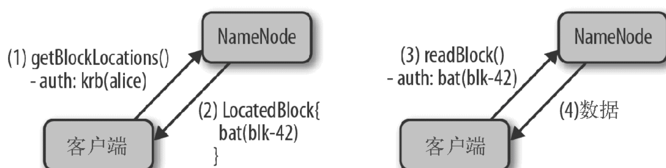


图 **5-2**：使用块访问令牌访问块

03. 作业令牌

提交一个 MapReduce 作业时，JobTracker 会创建一个名为“作业令牌”的密钥，该令牌被作业中的任务用于向 TaskTracker 发起身份验证。JobTracker 将作业令牌放置在 HDFS 中 JobTracker 的系统目录下，并通过 RPC 分发给各个 TaskTracker。TaskTracker 会将该令牌存放在本地磁盘的作业目录下，该目录仅能被作业用户访问。作业令牌用于给任务和 TaskTracker 之间的 RPC 通信进行身份验证，也用于生成一种散列值，该散列值确保通过 HTTP 传输的中间输出在 shuffle 阶段仅能被该作业的任务访问。此外，返回 shuffle 数据的 TaskTracker 会计算一个散列值，供每个任务验证它是在同真实的 TaskTracker——而不是伪装者——进行对话。

图 5-3 的时序图展示了作业建立阶段使用的身份验证方法。首先，客户端通过 Kerberos 认证请求建立一个新作业 (1)。JobTracker 给客户端响应作业 ID，用于该作业的唯一标识 (2)。客户端随之向 NameNode 请求委托令牌，并使用 JobTracker 作为更新器 (3)。NameNode 给客户端响应委托令牌 (4)。只有客户端通过 Kerberos 身份验证后，才会被分发委托令牌。最后，通过 Kerberos 与 JobTracker 进行身份验证的客户端将委托令牌与其他所需的作业细节发送给 JobTracker。

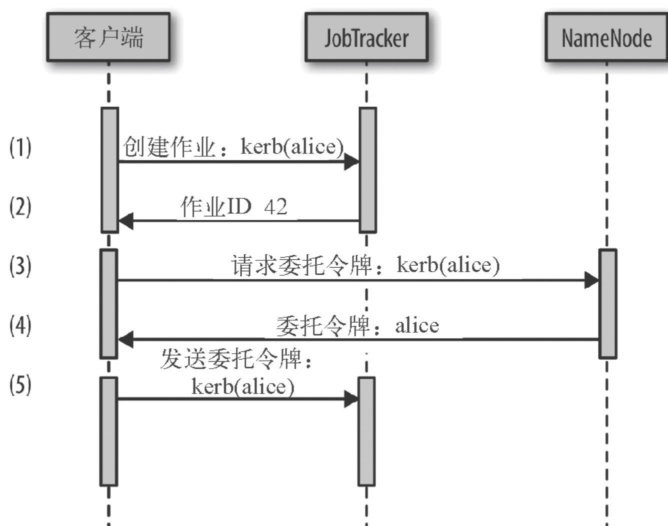


图 **5-3**：建立作业

开始执行作业时，情况会更加有意思，如图 5-4 所示。

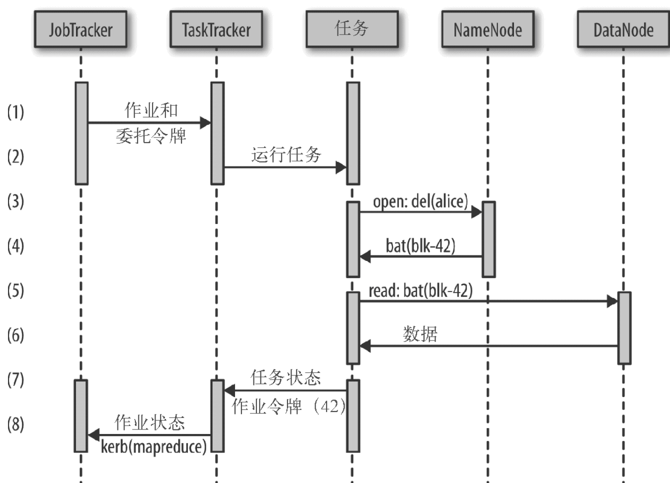


图 **5-4**：执行作业

JobTracker 会生成一个作业令牌，并将作业令牌、委托令牌和其他所需信息打包发送给 TaskTracker(1)。JobTracker 与 TaskTracker 对话时，需经过 Kerberos 身份验证。TaskTracker 会把接收到的令牌存储至一个仅能被提交作业的用户访问的目录，然后运行任务 (2)。任务通过委托令牌打开文件，并请求块地址以获得输入分片 (3)。NameNode 会将块地址和相应的块访问令牌响应给发起请求的任务 (4)。该任务随之使用获得的块访问令牌，从 DataNode 请求读取数据 (5)，DataNode 随之给予响应 (6)。作业执行过程中，任务会通过作业令牌进行身份验证，向 TaskTracker 报告任务状态 (7)。然后 TaskTracker 会通过 Kerberos 身份验证，并向 JobTracker 报告状态，从而收集整体作业状态 (8)。

5.2.4　用户模拟

Hadoop 生态系统中，有许多服务能代表终端用户执行事务。为了保证安全，这些服务必须对自己的客户端进行身份验证，并且能够被信任，以模拟其他用户。Oozie、Hive（HiveServer2 中）和 Hue 均支持，访问 HDFS、MapReduce、YARN 和 HBase 时模拟终端用户。安全模拟工作贯穿于这些服务，它们指定哪些用户可以执行模拟，以这种方式支持安全模拟。一个受信任的用户需要代表另一个用户执行操作时，她必须验证自己的身份，并提供其代替用户的用户名。受信任用户可以被限制为仅能模拟特定用户组，且为了进一步约束这些用户的特权，还可以限制他们仅能从某些主机访问 Hadoop。

模拟有时也称代理机制。能够执行模拟行为的用户（如能代表其他用户的）称为代理。启用模拟的配置参数为 `hadoop.proxyuser.<proxy>.hosts` 和 `hadoop.proxyuser.<proxy>.groups`，其中 `<proxy>` 是执行模拟行为用户的用户名。参数值为按逗号分隔、且顺序排列的主机和组列表，若为 `*` 则意味着所有主机 / 组。若用户同时需要 Hue 和 Oozie 的代理功能，但又需要限制 Oozie 能够代理的 `oozie-user` 组的成员，则需要使用示例 5-4 中的配置。

示例 5-4　代理的示例配置

```
<!-- Configure Hue impersonation from hue.example.com -->
<property>
  <name>hadoop.proxyuser.hue.hosts</name>
  <value>hue.example.com</value>
</property>
<property>
  <name>hadoop.proxyuser.hue.groups</name>
  <value>*</value>
</property>

<!--
    Configure Oozie impersonation from oozie01.example.com and
    oozie02.example.com for users in oozie-users
  -->
<property>
  <name>hadoop.proxyuser.oozie.hosts</name>
  <value>oozie01.example.com,oozie02.example.com</value>
</property>
<property>
  <name>hadoop.proxyuser.oozie.groups</name>
  <value>oozie-users</value>
</property>
```

5.2.5　配置

对于产品部署，Hadoop 支持使用 `Kerberos` 机制进行身份验证。按照 Kerberos 的身份验证机制配置时，所有用户和守护进程必须提供有效的凭证以访问 PRC 接口。这意味着，必须替集群中的每一个服务 / 守护进程对都创建一个 Kerberos 服务主体。回顾第 4 章讲过的服务主体名称（SPN）的概念，它由三部分组成：服务名称、主机名称、域。Hadoop 中，每个隶属于某服务的守护进程都会使用该服务的名称（HDFS 服务用 `hdfs`、MapReduce 服务用 `mapred`、YARN 服务用 `yarn`）。另外，若要为多个 Web 接口启用 Kerberos 身份验证机制，还需要向主体提供 `HTTP` 服务的名称。

下面了解如何配置一个使用 Kerberos 身份验证机制的集群。假设有一个集群，包含表 5-5 所示的主机和服务。

> 表 5-5 展示的服务层仅用作示例，并不是配置集群的最好选择。对于初学者，我们使用 YARN 和 MR1 服务的配置展示示例。这仅意味着，给出全方位的配置示例需要用到这两个服务（并不是说用户必须使用这两个服务）。实际部署中，可能只部署一个或另一个。与之类似，如果像在本例中使用 HA 运行两个 NameNode，也不需要再另外部署一个 SecondaryNameNode。重申一下，这仅仅是为了让我们的示例更全面而详尽。

表**5-5**：服务层

主机	守护进程
nn1.example.com	NameNode JournalNode
nn2.example.com	NameNode JournalNode
snn.example.com	SecondaryNameNode JournalNode
rm.example.com	ResourceManager
jt.example.com	JobTracker JobHistoryServer
dn1.example.com	DataNode TaskTracker NodeManager
dn2.example.com	DataNode TaskTracker NodeManager
dn3.example.com	DataNode TaskTracker NodeManager

第一步，要在 Kerberos KDC 中创建所需的 SPN，并为每台服务器上的每一个守护进程导出 keytab（密钥库）文件。每个主机／角色所需 SPN、它们各自所属的 keytab 文件的推荐名称如表 5-6 所示。对于每个服务器需要创建不同的 keytab 文件。为了能在所有主机上使用相同配置（此时不同服务器上的同名 keytab 文件含有不同密钥），建议每个守护进程都使用一致的名称。

表**5-6**：所需**Kerberos**规则

			密钥表路径
NameNode	nn1.example.com	nn/nn1.example.com@EXAMPLE.COM HTTP/nn1.example.com@EXAMPLE.COM	hdfs.keytab
NameNode	nn2.example.com	nn/nn2.example.com@EXAMPLE.COM HTTP/nn2.example.com@EXAMPLE.COM	hdfs.keytab
SecondaryNameNode	snn.example.com	hdfs/snn.example.com@EXAMPLE.COM HTTP/snn.example.com@EXAMPLE.COM	hdfs.keytab
ResourceManager	[illegible]	[illegible]r.com@EXAMPLE.COM	yarn.keytab
JobTracker	jt.example.com	mapred/jt.example.com@EXAMPLE.COM HTTP/jt.example.com@EXAMPLE.COM	mapred.keytab
JobHistoryServer	[illegible]	[illegible]ple.com@EXAMPLE.COM	mapred.keytab
DataNode	dn1.example.com	hdfs/dn1.example.com@EXAMPLE.COM HTTP/dn1.example.com@EXAMPLE.COM	hdfs.keytab
TaskTracker	dn1.example.com	mapred/dn1.example.com@EXAMPLE.COM HTTP/dn1.example.com@EXAMPLE.COM	mapred.keytab
NodeManager	dn1.example.com	yarn/dn1.example.com@EXAMPLE.COM HTTP/dn1.example.com@EXAMPLE.COM	yarn.keytab
DataNode	dn2.example.com	hdfs/dn2.example.com@EXAMPLE.COM HTTP/dn2.example.com@EXAMPLE.COM	hdfs.keytab
TaskTracker	dn2.example.com	mapred/dn2.example.com@EXAMPLE.COM HTTP/dn2.example.com@EXAMPLE.COM	mapred.keytab
NodeManager	dn2.example.com	yarn/dn2.example.com@EXAMPLE.COM HTTP/dn2.example.com@EXAMPLE.COM	yarn.keytab
DataNode	dn3.example.com	hdfs/dn3.example.com@EXAMPLE.COM HTTP/dn3.example.com@EXAMPLE.COM	hdfs.keytab
TaskTracker	dn3.example.com	mapred/dn3.example.com@EXAMPLE.COM HTTP/dn3.example.com@EXAMPLE.COM	mapred.keytab
NodeManager	dn3.example.com	yarn/dn3.example.com@EXAMPLE.COM HTTP/dn3.example.com@EXAMPLE.COM	yarn.keytab

 导出 keytab 文件时需要注意，因为默认的设置是，每导出一个主体就将 Kerberos 密钥随机化。用户可以将每个主体导出，然后使用 ktutil 工具将每个守护进程的密钥和 keytab 文件结合起来。

我们推荐将相应的 keytab 文件存储在 $HADOOP_CONF_DIR 目录中（一般是 /etc/hadoop/conf）。

 配置文件完整示例

配置文件的完整示例在本书 GitHub 主页的示例资源（http://bit.ly/1EAEmTr）中。

创建所需的 SPN 并分发 keytab 后，需要对 Hadoop 进行配置，以使用 Kerberos 进行身份验证。如示例 5-5 所示，首先将 core-site.xml 文件的 hadoop.security.authentication 参数设为 kerberos。

示例 **5-5**　将身份验证类型配置为 Kerberos

```
<property>
  <name>hadoop.security.authentication</name>
  <value>kerberos</value>
</property>
```

01. **HDFS**

接下来，需要为每个服务配置 Kerberos 主体和 keytab 文件。对于 NameNode，还必须设置 `dfs.block.access.token.enable` 为 `true`，以启用块访问令牌。NameNode 的配置应当在 `hdfs-site.xml` 文件中进行，如示例 5-6 所示。

示例 **5-6**　配置 NameNode

```
<property>
  <name>dfs.block.access.token.enable</name>
  <value>true</value>
</property>
<property>
  <name>dfs.namenode.keytab.file</name>
  <value>hdfs.keytab</value>
</property>
<property>
  <name>dfs.namenode.kerberos.principal</name>
  <value>hdfs/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>dfs.namenode.kerberos.internal.spnego.principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
```

如果没有启用 HDFS 的高可用模式，接下来则需要在 `hdfs-site.xml` 文件中对 SecondaryNameNode 进行配置，如示例 5-7 所示。

示例 **5-7**　配置 SecondaryNameNode

```
<property>
  <name>dfs.secondary.namenode.keytab.file</name>
  <value>hdfs.keytab</value>
</property>
<property>
```

```
  <name>dfs.secondary.namenode.kerberos.principal</name>
  <value>hdfs/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>dfs.secondary.namenode.kerberos.internal.spnego.principal</n
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
```

如果启用了 HDFS 的高可用模式，则需要在 `hdfs-site.xml` 文件中对 JournalNode 进行如下配置，如示例 5-8 所示。

示例 5-8　配置 JournalNode

```
<property>
  <name>dfs.journalnode.keytab.file</name>
  <value>hdfs.keytab</value>
</property>
<property>
  <name>dfs.journalnode.kerberos.principal</name>
  <value>hdfs/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>dfs.journalnode.kerberos.internal.spnego.principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
```

接下来，按照如下设置，在 `hdfs-site.xml` 文件中配置 DataNode。除了要配置 keytab 和主体名之外，还必须对 DataNode 进行设置，使其对 RPC 和 HTTP 服务使用特权端口。之所以使用特权端口是因为，DataNode 的数据传输协议并不使用 Hadoop 的 RPC 框架。通过特权端口，DataNode 能够验证其是由 `root` 通过 `jsvc` 发起的，如示例 5-9 所示。

示例 5-9　配置 DataNode

```
<property>
  <name>dfs.datanode.address</name>
  <value>0.0.0.0:1004</value>
</property>
<property>
  <name>dfs.datanode.http.address</name>
  <value>0.0.0.0:1006</value>
  </property>
<property>
  <name>dfs.datanode.keytab.file</name>
  <value>hdfs.keytab</value>
```

```
</property>
<property>
  <name>dfs.datanode.kerberos.principal</name>
  <value>hdfs/_HOST@EXAMPLE.COM</value>
</property>
```

WebHDFS 是基于 REST 的数据访问协议。WebHDFS 的职责是，从存有被读取块的 DataNode 中读取数据，并通过 HTTP 对外提供数据。为了安全访问 WebHDFS，需要设置存于 NameNode 和 DataNode 的 `hdfs-site.xml` 文件中的下列参数，如示例 5-10 所示。

示例 **5-10**　配置 WebHDFS

```
<property>
  <name>dfs.web.authentication.kerberos.keytab</name>
  <value>hdfs.keytab</value>
</property>
<property>
  <name>dfs.web.authentication.kerberos.principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
```

现在，HDFS 的配置就算完成了！

02. **YARN**

现在配置 YARN，从 ResourceManager 开始。先设置 `yarn-site.xml` 文件中的配置参数，如示例 5-11 所示。

示例 **5-11**　配置 ResourceManager

```
<property>
  <name>yarn.resourcemanager.principal</name>
  <value>yarn/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>yarn.resourcemanager.webapp.spnego-principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>yarn.resourcemanager.keytab</name>
  <value>yarn.keytab</value>
</property>
<property>
```

```
  <name>yarn.resourcemanager.webapp.spnego-keytab-file</name>
  <value>yarn.keytab</value>
</property>
```

在 yarn-site.xml 文件中对参数进行设置，将 NodeManager 配置为使用 Kerberos，如示例 5-12 所示。

示例 **5-12**　配置 NodeManager

```
<property>
  <name>yarn.nodemanager.principal</name>
  <value>yarn/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>yarn.nodemanager.webapp.spnego-principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>yarn.nodemanager.keytab</name>
  <value>yarn.keytab</value>
</property>
<property>
  <name>yarn.nodemanager.webapp.spnego-keytab-file</name>
  <value>yarn.keytab</value>
</property>
```

除了将 NodeManager 配置为使用 Kerberos 进行身份验证外，还需要设置 NodeManager 使用 `LinuxContainerExecutor`。`LinuxContainerExecutor` 使用一个 setuid 程序启动 YARN 容器，这使得 NodeManager 能够使用提交作业的用户 UID 执行容器。因此，需要一个安全的配置，以确保 Alice 不能访问 Bob 执行的容器所创建的文件。如果没有 `LinuxContainerExecutor`，`yarn` 用户和容器就能够相互访问彼此的本地文件，这会造成所有容器都被运行。首先，配置 yarn-site.xml 文件，如示例 5-13 所示。

示例 **5-13**　配置 NodeManager 使用 LinuxContainerExecutor

```
<property>
  <name>yarn.nodemanager.container-executor.class</name>
  <value>org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExe
</property>
<property>
  <name>yarn.nodemanager.linux-container-executor.group</name>
```

```
  <value>yarn</value>
</property>
```

还需要配置执行器程序本身，配置 container-executor.cfg 文件中的参数即可，如示例 5-14 所示。yarn.nodemanager.linux-container-executor.group 参数的值应当与 yarn-site.xml 文件和 container-executor.cfg 文件中的值保持一致。通常该值被设为 yarn。

示例 **5-14**　配置 LinuxContainerExecutor

```
yarn.nodemanager.linux-container-executor.group=yarn
min.user.id=1000
allowed.system.users=nobody,impala,hive,llama
banned.users=root,hdfs,yarn,mapred,bin
```

对 min.user.id 的设定用于防止 LinuxContainerExecutor 运行 UID 小于该参数值的容器。该值通常被设为 1000 或 500，具体取决于普通用户的 UID 在实际环境中的开始位置。除了该设置之外，还可以设置明确的用户白名单和黑名单。这种设置专门允许用户中的 **hive** 用户运行容器。启用 Apache Sentry 时需要进行该项配置，因为 Sentry 启用时，Hive 代理是被关闭的。

配置 YARN 使用 Kerberos 的最后一步是配置 JobHistoryServer，设定 mapredsite.xml 文件中的参数即可，如示例 5-15 所示。

示例 **5-15**　为 Kerberos 配置 JobHistoryServer

```
<property>
  <name>mapreduce.jobhistory.principal</name>
  <value>mapred/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>mapreduce.jobhistory.webapp.spnego-principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>mapreduce.jobhistory.keytab</name>
  <value>mapred.keytab</value>
```

```
</property>
<property>
  <name>mapreduce.jobhistory.webapp.spnego-keytab-file</name>
  <value>mapred.keytab</value>
</property>
```

03. **MapReduce (MR1)**

如果用户仍在使用 MR1，则可以跳过上面的 YARN 配置，直接对 JobTracker 和 TaskTracker 进行配置。首先，配置 `mapred-site.xml` 文件中的参数，如示例 5-16 所示。

示例 **5-16**　为 Kerberos 配置 JobTracker

```
<property>
  <name>mapreduce.jobtracker.kerberos.principal</name>
  <value>mapred/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>mapreduce.jobtracker.keytab.file</name>
  <value>mapred.keytab</value>
</property>
```

对 TaskTracker 的配置也同样简单。配置 `mapred-site.xml` 文件中的参数，如示例 5-17 所示。

示例 **5-17**　为 Kerberos 配置 TaskTracker

```
<property>
  <name>mapreduce.tasktracker.kerberos.principal</name>
  <value>mapred/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>mapreduce.tasktracker.keytab.file</name>
  <value>mapred.keytab</value>
</property>
```

示例 5-12 和示例 5-13 中，配置 NodeManager 时也需要启用 `LinuxContainerExecutor`。这相当于 MR1 中的 `LinuxTaskController`。先配置 `mapred-site.xml` 文件中的参数，如示例 5-18 所示。

示例 **5-18**　使用 LinuxTaskController 配置

TaskTracker

```
<property>
  <name>mapred.task.tracker.task-controller</name>
  <value>org.apache.hadoop.mapred.LinuxTaskController</value>
</property>
<property>
  <name>mapreduce.tasktracker.group</name>
  <value>mapred</value>
</property>
```

还需要配置任务控制器本身。设置 `taskcontroller.cfg` 文件中的参数，确保 `mapreduce.tasktracker.group` 的值与 `mapred-site.xml` 文件中使用的值相匹配，该值通常为 `mapred`。与 `LinuxContainerExecutor` 不同，`LinuxTaskController` 不允许设置一个被允许的系统用户列表。这意味着，如果你需要允许特定的系统用户运行作业，那么可能要降低 `min.user.id` 的值，并增加 `banned.users` 列表中明确禁止的用户个数，如示例 5-19 所示。

示例 **5-19**　配置 LinuxTaskController

```
mapred.local.dir=/mapred/local
hadoop.log.dir=/var/log/hadoop-0.20-mapreduce
mapreduce.tasktracker.group=mapred
banned.users=root,mapred,hdfs,bin
min.user.id=1000
```

04. **Oozie**

如前所述，Oozie 支持使用 Kerberos 进行身份验证。在 Oozie 中启用身份验证前，需要先配置 Oozie，使其在访问 Hadoop 时对自己进行验证。可以通过配置如下参数实现（示例 5-20 展示了一种合适的参数配置）。

`oozie.service.HadoopAccessorService.kerberos.enable`

　　`hadoop.security.authentication` 设为 `kerberos` 时，该值设为 `true`。

`local.realm`

将该值设为 Hadoop 集群的默认域，这与 krb5.conf 文件中 default_realm 配置的域应该是相同的。

oozie.service.HadoopAccessorService.kerberos.princi

Oozie 用于进行身份验证的 Kerberos 主体，通常是 oozie/<fqdn>@<REALM>，其中 <fqdn> 是运行 Oozie 的服务器的完全限定域名（即域名全称），<REALM> 是本地 Kerberos 域。

oozie.service.HadoopAccessorService.keytab.file

keytab 文件路径，该文件含有配置的 Kerberos 主体的密钥。

示例 **5-20**　配置 Oozie，使之支持启用 Kerberos 的 Hadoop 集群

```
<property>
  <name>oozie.service.HadoopAccessorService.kerberos.enabled</name>
  <value>true</value>
</property>
<property>
  <name>local.realm</name>
  <value>EXAMPLE.COM</value>
</property>
<property>
  <name>oozie.service.HadoopAccessorService.keytab.file</name>
  <value>oozie.keytab</value>
</property>
<property>
  <name>oozie.service.HadoopAccessorService.kerberos.principal</name
  <value>oozie/oozie01.example.com@EXAMPLE.COM</value>
</property>
```

Oozie 被配置为支持启用 Kerberos 的 Hadoop 集群后，即可配置 Oozie 使用 Kerberos 进行用户身份验证。相关设置如下（示例 5-21 展示了一个配置样例）。

oozie.authentication.type

设置用户所需的认证类型，可设为 simple（默认值）、kerberos 或实现 Hadoop AuthenticationHandler 接口的类的全称。

`oozie.authentication.token.validity`

认证令牌的有效时间，以秒为单位。认证令牌会在初始的认证方法（通常是 Kerberos/SPNEGO）后，以 cookie 的形式返回。

`oozie.authentication.signature.secret`

用于对认证令牌签名的密文。如果为空，则会在启动时生成一个随机密文。如果 Oozie 被配置为 HA 模式，则所有 Oozie 服务器上的该值必须相同。

`oozie.authentication.cookie.domain`

生成认证 cookie 时使用的域名，该值应当设置为集群的域名。

`oozie.authentication.kerberos.principal`

用于 Oozie 服务的 Kerberos 主体。由于 Oozie 使用 SPNEGO over HTTP 进行认证，该值必须设为 `HTTP/<fqdn>@<REALM>`，其中 `<fqdn>` 是 Oozie 服务器的域名全称，`<REALM>` 是本地 Kerberos 域。

`oozie.authentication.kerberos.keytab`

keytab 文件路径，该文件包含 Kerberos 主体的密钥。

`oozie.authentication.kerberos.name.rules`

将 Kerberos 主体转换为本地用户名的规则。该参数的格式与 Hadoop 的 `hadoop.security.auth_to_local` 参数的格式一样，如何进行配置请参考 5.1.1 节和示例 5-21。

示例 **5-21**　配置使用 Kerberos 认证的 Oozie

```
<property>
  <name>oozie.authentication.type</name>
  <value>kerberos</value>
</property>
<property>
  <name>oozie.authentication.token.validity</name>
  <value>36000</value>
</property>
```

```
<property>
  <name>oozie.authentication.signature.secret</name>
  <value>FiSEcve7lBsdGpvr</value>
</property>
<property>
  <name>oozie.authentication.cookie.domain</name>
  <value>example.com</value>
</property>
<property>
  <name>oozie.authentication.kerberos.principal</name>
  <value>HTTP/oozie01.example.com@EXAMPLE.COM</value>
</property>
<property>
  <name>oozie.authentication.kerberos.principal</name>
  <value>oozie.keytab</value>
</property>
<property>
  <name>oozie.authentication.kerberos.name.rules</name>
  <value>DEFAULT</value>
</property>
```

如果以 HA 模式运行 Oozie，则需要一些额外配置。首先应当配置 `oozie-site.xml` 文件中的 `oozie.zookeeper.secure`，让 Oozie 使用 ZooKeeper ACL，如示例 5-22 所示。

示例 **5-22**　在 oozie-site.xml 中为 Oozie 配置 ZooKeeper ACL

```
<property>
  <name>oozie.zookeeper.secure</name>
  <value>true</value>
</property>
```

如果在 Hadoop 2.5.0 以前的版本中使用 Oozie，则需要在 HTTP 主体名称中使用负载均衡器的完全限定域名。例如，如果有运行在 `oozie01.example.com` 和 `oozie02.example.com` 上的 Oozie 服务器，以及运行在 `oozie.example.com` 上的负载均衡器，那么应当在所有 Oozie 服务器中使用 `HTTP/oozie.example.com@EXAMPLE.COM` 作为主体。该模式下，只能通过负载均衡器访问。另外，一些诸如日志流的功能也不能用。这一步应当在 `oozie-site.xml` 文件中配置以下内容，如示例 5-23 所示。

示例 5-23　在负载均衡环境下配置 Oozie SPN

```
<property>
  <name>oozie.authentication.kerberos.principal</name>
  <value>HTTP/oozie.example.com@EXAMPLE.COM</value>
</property>
```

从 Hadoop 2.5.0 开始，可以在 Oozie 的 keytab 文件中包含多个 Kerberos 主体。这种情况下，可以在 keytab 文件中加入负载均衡器的主体和特定服务器的主体（例如 `HTTP/oozie.example.com@EXAMPLE.COM` 和 `HTTP/oozie01.example.com@EXAMPLE.COM`）。随后需要将 `oozie.authentication.kerberos.principal` 设为 `*`，如示例 5-23 所示。

示例 5-24　配置具有多个 SPN 的 Oozie

```
<property>
  <name>oozie.authentication.kerberos.principal</name>
  <value>*</value>
</property>
```

05. **HBase**

配置使用 Kerberos 认证的 HBase 与配置核心 Hadoop 是非常类似的。为了节约篇幅，建议参考 *The Apache HBase Reference Guide*（http://hbase.apache.org/book.html）的 Securing Apache HBase（http://hbase.apache.org/book.html#security）章节。

5.3　小结

本章介绍了身份的概念，并展示了 Hadoop 如何使用 Kerberos 主体名映射用户名。我们还看到 Hadoop 会获取用户的组成员信息，这一点在第 6 章讲述授权时会很重要。

我们还分析了集群中进行身份验证的不同方式。虽然 Kerberos 是一种经典的例子，而且也很常用，但我们也看到，还有其他使用委托令牌进行身份验证的方式。这是 Hadoop 身份验证架构中的一个关键部分，因为它减少了完成工作流所必需的 Kerberos 认证路径的数量

（例如，一个执行 Hive 查询 Oozie 工作流转化为 MapReduce 作业其实就是处理文件）。如果没有委托令牌，那么每个步骤都要请求 Kerberos 服务票据，这增加了 Kerberos KDC 的压力。

最后介绍了用户模拟的概念，讨论了系统用户如何能够代表其他用户进行身份验证。这是个常用概念，因为终端用户经常使用介于他们和他们尝试访问的服务之间的工具。使用用户模拟，系统或服务就可以使用另外一个远程服务进行身份验证，然后像终端用户直接验证一样获取访问权限。

接下来，我们将走近能够访问数据和服务的用户，接着身份验证、授权和审计的话题讨论授权。

第 6 章　授权

通过 5.2 节，我们了解了各种 Hadoop 生态系统项目如何支持强认证以确保用户就是他们所声称的那个人。然而，身份验证只是安全整体的一部分，你还需要对已认证用户可以执行哪些操作和访问哪些数据进行建模。这种资源保护的方式称为授权，这可能也是Hadoop安全中最复杂的话题之一。每个服务提供的服务是相对独特的，因此它们支持的授权模型也相对独特。本章将按照每个服务如何实现授权分成若干小节。

我们先看 HDFS 及其对 POSIX 文件权限的支持，以及对利用服务层授权限制用户访问特定 HDFS 功能的支持。接下来讲解 MapReduce 和 YARN，它们支持的服务层授权和用于控制系统资源访问的队列模型是类似的。对于 MapReduce 和YARN，授权有助于安全和资源管理／多任务处理（关于资源管理的更多内容，推荐阅读 Eric Sammer 的《Hadoop 技术详解》）。最后介绍主流的 BigTable、Apache HBase 以及 Apache Accumulo 的授权特性，并分别讨论基于角色和基于属性的安全策略的优缺点，以及对单元级安全和列级安全进行比较。

6.1　HDFS授权

HDFS 中，每进行一次文件或目录访问都必须先经过授权检查。HDFS 采用兼容 POSIX 的文件系统中常用的授权方式。权限由三种不同类型的用户控制：**owner**（所有者）、**grou**（组）和 **others**（其他）。每个文件或目录都归一个特定的用户所有，该用户构成了这个对象的 `owner` 类。对象还被分配了一个组，该组的所有成员构成了该对象的 `group` 类。不是对象所有者、也不属于被分配到该对象组的所有其他用户则构成 `others` 类。读、写和执行权限可被独立地授予各个类。

这些权限由一个对权限值（4 代表读，2 代表写，1 代表执行）求和后的八进制数字表示。比如要表示一个类具有一个目录的读和执行权限，则需要分配其一个八进制数字 5（4+1）。

HDFS 中，将执行权限分配给文件虽然不是非法的，但也没有意义。对于目录来说，执行位允许在知道文件名的前提下访问文件内容和元数据信息。为了列出一个目录里的所有文件名，需要该目录的读权

限。

无论文件或目录的权限是什么，运行 NameNode 的用户（通常是 hdfs）以及 dfs.permissions.supergroup 中定义的组（默认情况下是 supergroup）中的成员，都可以对任意文件和目录进行读、写和删除操作。就 HDFS 而言，他们相当于 Linux 系统中的 root 用户。

分配给 owner、group 和 others 的权限可以用依次连接的 3 个八进制值表示。例如有一个文件，owner 对其具有读和写权限，而所有其他用户对其只有读权限，那么该文件的权限就可以表示为 644。6 是分配给 owner 类的，因为其具有读和写权限（4+2）；4 是分配给 group 和 others 类的，因为他们只有读权限。对于一个所有用户都对其拥有所有权限的文件，其权限为 777。

除了标准权限之外，HDFS 还支持三种额外的特殊权限：setuid、setgid 和 sticky。这些权限也是用八进制数字表示的，4 代表 setuid，2 代表 setgid，1 代表 sticky。这些权限是可选的，如果使用，那么它们会包含在常规权限位的左侧。由于 HDFS 中的文件是无法执行的，因此 setuid 没有作用。setgid 同样对文件没有影响，但对于目录而言，它将新建的子文件和子目录的 group 强制设为其父节点的 group。这在 HDFS 中是默认行为，因此也没有必要刻意对目录启用 setgid。最后一个权限通常称为粘滞位，它意味着目录中的文件只能被该文件的所有者删除。没有设置粘滞位的情况下，文件可以被具有其所属目录写权限的任意用户删除。HDFS 中，无论有没有设置粘滞位，目录所有者以及 HDFS 超级管理员都可以对文件进行删除。粘滞位对于目录很有用，例如对于 /tmp 目录，你需要所有用户都拥有对该目录的写权限，但该目录中的数据只能被各自的所有者删除。

HDFS扩展ACL

使用 owner、group 和 world（全局）的基本 POSIX 权限，以允许访问给定的文件和目录，往往不那么简单。两个或更多不同组的用户需要访问同样的 HDFS 目录时，会怎样？如果使用基本 POSIX 权限，那么管理员有两种方案 :(1) 将该目录设为可全局访问；(2) 创建一个包含所有需要访问该目录的用户组，并分配组权限。这样做并不理想，因为方案 (1) 使得数据能够被预期之外的用户访问，而方案 (2) 从组管理的角度来说，是个很头疼的工作。当一个用户组需要读权限，而另外一个用户组需要读和写权限时，这个问题就会变得更加复杂。

随着 Hadoop 2.4 的发布，HDFS 现在配备了扩展 ACL，这些 ACL 的工作方式与 Unix 环境中扩展 ACL 的工作方式几乎一样。它允许 HDFS 中的文件和目录拥有比基本 POSIX 权限更多的权限。

要使用 HDFS 的扩展 ACL，必须先在 NameNode 上启用：将 `hdfs-site.xml` 中的 `dfs.namenode.acls.enabled` 属性设为 `true`。示例 6-1 展示了如何使用 HDFS 的扩展 ACL。

示例 **6-1**　HDFS 扩展 ACL

```
[alice@hadoop01 ~]$ hdfs dfs -ls /data
Found 1 items
drwxr-xr-x   - alice analysts          0 2014-10-25 19:03 /data/alice
[alice@hadoop01 ~]$ hdfs dfs -getfacl /data/alice
# file: /data/alice
# owner: alice
# group: analysts
user::rwx
group::r-x
other::r-x
[alice@hadoop01 ~]$ hdfs dfs -setfacl -m user:bob:r-x /data/alice
[alice@hadoop01 ~]$ hdfs dfs -setfacl -m group:developers:rwx /data/alice
[alice@hadoop01 ~]$ hdfs dfs -ls /data
Found 1 items
drwxr-xr-x+   - alice analysts          0 2014-10-25 19:03 /data/alice
[alice@hadoop01 ~]$ hdfs dfs -getfacl /data/alice
# file: /data/alice
# owner: alice
# group: analysts
user::rwx
user:bob:r-x
group::r-x
group:developers:rwx
mask::rwx
other::r-x
[alice@hadoop01 ~]$ hdfs dfs -chmod 750 /data/alice
[alice@hadoop01 ~]$ hdfs dfs -getfacl /data/alice
# file: /data/alice
# owner: alice
# group: analysts
user::rwx
group::r-x
group:developers:rwx    #effective:r-x
mask::r-x
other::---
[alice@hadoop01 ~]$ hdfs dfs -setfacl -b /data/alice
[alice@hadoop01 ~]$ hdfs dfs -getfacl /data/alice
# file: /data/alice
# owner: alice
# group: analysts
```

```
user::rwx
group::r-x
other::---
```

这里有几点值得强调。首先，默认情况下，文件和目录没有任何ACL。添加 ACL 项到一个对象后，HDFS 列表会在权限列表中加入一个 +，例如 drwxr-xr-x+。另外，添加 ACL 项后，一个叫作 mask（掩码）的属性会列在 ACL 里，它定义了最严权限。比如用户 Bob 拥有 rwx 权限，但 mask 是 r-x，则 Bob 的有效权限是 r-x。getfacl 的输出里也会这样注明，如示例 6-1 所示。

关于 mask 的另一个重要的地方是，它会根据 ACL 中设置的最小限制权限做出调整。例如一个 mask 目前是 r-x，为一个组加入一条新的 ACL 项以授予其 rwx 权限时，mask 就会被调整为 rwx。

> 在包含扩展 ACL 的文件或目录上，设置标准 POSIX 权限可能会立即对所有 ACL 项产生影响，因为 hdfs dfs -chmod 实际上会设置 mask，无论当前的 ACL 项是什么。例如，在文件或目录上设置 700 权限的结果是，除所有者之外，对所有 ACL 项的实际权限都是禁止访问！

例子中最后一部分展示了如何彻底移除目录的 ACL 项，只保留基本 POSIX 权限。关于扩展 ACL 的最后一点是，对于每个对象（如文件或目录），最多只能有 32 个 ACL 项。也就是说，除去 4 个项被 user、group、other 和 mask 占据，还可以添加 28 个项。超出之后，NameNode 就会抛出一个异常：setfacl: Invalid ACL: ACL has 33 entries, which exceeds maximum of 32。

另一个扩展 ACL 中有用的特性是默认 **ACL** 的使用。一条默认 ACL 只能应用到一个目录，效果是，所有在该目录中创建的子目录和子文件都会继承父目录的默认 ACL。例如，一个目录具有一条默认 ACL 项：default:group:analysts:rwx，那么该目录中创建的所有文件都会具有 group:analysts:rwx 项，而子目录会同时具有复制的默认 ACL 和访问 ACL。要设置默认 ACL，只需在 setfacl 命令中，于用户或组的 ACL 项前加入 default: 即可。记住，默认 ACL 自身并不实施授权，它只定义了新创建子目录和文件的继承行为。

6.2　服务级授权

Hadoop 还支持在服务级进行授权。这可以用于控制哪些用户或用户组可以访问特定协议，也可以用于阻止流氓进程伪装成守护进程。将 `core-site.xml` 中的 `hadoop.security.authorization` 项设为 `true`，即可启用服务级授权。实际的策略在 `hadoop-policy.xml` 文件中进行配置，这个文件在结构上与标准配置文件类似，每个属性用一个 `property` 标签定义，每个 `property` 标签包含一个表示属性名称和一个表示属性值的子标签。每个服务级授权属性使用一组以逗号分隔的、能够访问该协议的用户和组，以定义一个访问控制列表（ACL）。两种列表使用空格分隔，开头的空格表示空的用户列表，结尾的空格表示空的组列表。特殊值 `*` 表示允许所有用户访问该协议（这也是默认设置）。表 6-1 提供了一些示例 ACL。

表6-1：Hadoop访问控制列表

ACL	含义
`*`	允许所有用户
` `	不允许任何用户
`alice,bob hdusers`	允许 alice、bob 和 hdusers 组里的任意用户
`alice,bob `	允许 alice 和 bob（末尾有空格），但不允许任何组
` hdusers`	允许 hdusers（开头有空格）组里的任意用户，但不允许其他用户

了解可用的 ACL 之前，我们先定义一些用户和组，以帮助指导配置。假定有一个含有少量用户和一个 Hadoop 管理员的小集群。集群用户拥有 Linux 工作站，我们想保证用户能够尽可能多地在他们的工作站上进行开发，因此不打算在工作站网络和集群之间放置防火墙。此外，假定还有一个定义了整个企业网用户和组的中心 Active Directory，集群的 KDC 被配置为使用单向信任，以允许 AD 用户不需要新凭证就能登录到集群。现在，我们想让 Hadoop 开发者能够访问集群，但又不想让整个公司都能浏览 HDFS 或者启动 MapReduce 作业。为了帮助设置，我们配置了两个组，一个为 `hadoop-users`，另一个为 `hadoop-admins`。由于这是一个新环境，我们首先只把 3 个用户——Alice、Bob 和 Joey 添到 `hadoop-users` 组。Joey 是认证过的 Hadoop 管理员，所以也将他加入 `hadoop-admins` 组。

支持服务级授权的有 HDFS、MapReduce（MR1）和 YARN（MR2）。示例中的协议 ACL 和推荐的配置值见表 6-2~表 6-4，它们分别为 HDFS、MapReduce（MR1）和 YARN（MR2）而定义。其中一些属性是服务间共享的（比如刷新策略配置的协议），

因此它们会在多个表中出现。由于 Hadoop 2.3 不包含 MR1，所以一些针对 MR1 策略的属性名会有所不同，部署 Hadoop 1.2 或者包含配合 HDFS 使用的 MR1 的 Hadoop 2.x 发行版时使用 MR1 属性名。

表6-2：**HDFS**服务级授权属性

属性名	描述
`security.client.protocol.acl`	访问 NameNode 协议的客户端，用户代码通过 `DistributedFileSystem` 类使用
`security.client.datanode.protocol.acl`	访问 DataNode 协议的客户端
`security.get.user.mappings.protocol.acl`	获取用户映射到的组的协议
`security.datanode.protocol.acl`	访问 NameNode 协议的 DataNode
`security.inter.datanode.protocol.acl`	访问 DataNode 协议的 DataNode
`security.namenode.protocol.acl`	访问 NameNode 协议的 Secondary-NameNode
`security.qjournal.service.protocol.acl`	访问 JournalNode 协议的 NameNode
`security.zkfc.protocol.acl`	"zkfc" 暴露的协议
`security.ha.service.protocol.acl`	"hdfs haadmin" 命令管理 NameNode 的 HA 状态所使用的协议
`security.refresh.policy.protocol.acl`	"hadoop dfsadmin" 命令加载最新 hadoop-policy.xml 文件所使用的协议
`security.refresh.user.mappings.protocol.acl`	刷新用户到组映射关系的协议

表6-3：**MapReduce (MR1)** 服务级授权属性

属性名	描述
`security.task.umbilical.protocol.acl`	MR 任务报告任务进度所用的协议注意：必须设为 `*`
`security.job.submission.protocol.acl`	客户端向 JobTracker 提交作业所用的协议
`security.inter.tracker.protocol.acl`	TaskTracker 与 JobTracker 通信所用的协议
`security.refresh.policy.protocol.acl`	"hadoop mradmin" 命令加载最新的 hadoop-policy.xml 文件所用的协议
`security.refresh.usertogroups.mappings.protocol.acl`	刷新用户到组映射关系的协议 注意：属性名在 Hadoop 2.0 中有所改变
`security.admin.operations.protocol.acl`	"hadoop mradmin" 命令刷新 JobTracker 的队列和节点所用的协议

表 6-4：**YARN**和**MR2**的服务级授权属性

属性名	描述
`security.job.task.protocol.acl`	MR 任务报告任务进度所用的协议注意：必须设为 `*`
`security.containermanagement.protocol.acl`	ApplicationMaster 与 NodeManager 通信所用的协议 注意：必须设为 `*`
`security.applicationmaster.protocol.acl`	ApplicationMaster 与 ResourceManager 通信所用的协议 注意：必须设为 `*`
`security.get.user.mappings.protocol.acl`	获取用户映射到的组的协议
`security.applicationclient.protocol.acl`	客户端向 ResourceManager 提交应用所用的协议
`security.job.client.protocol.acl`	作业客户端与 MR ApplicationMaster 通信所用的协议
`security.mrhs.client.protocol.acl`	作业客户端与 MapReduce JobHistory Server 通信所用的协议
`security.resourcetracker.protocol.acl`	访问 NodeManager 协议的 ResourceManager
`security.resourcemanager-administration.protocol.acl`	"yarn rmadmin" 命令管理 ResourceManager 所用的协议
`security.resourcelocalizer.protocol.acl`	ResourceLocalizationService 和 NodeManager 之间通信所用的协议
`security.ha.service.protocol.acl`	"yarn rmadmin" 命令管理 ResourceManager 的 HA 状态所用的协议

命令加载最新的 hadoop-policy.xml 文件所用的协议		
刷新用户到组映射关系的协议	...mappings.protocol.acl	

你可能会注意到，虽然我们想让集群尽量封闭，但还是不得不配置了 4 个允许任意用户连接的协议。这么做的原因是，这些协议需要被正在运行的任务访问，这些任务会使用应用尝试（Application Attempt）或任务尝试（Task Attempt）的身份。而任务每次运行使用的身份都不一样，也与运行作业的用户名毫无关系。由于这些身份无法提前枚举，所以也无法将它们列到 ACL 或者添加到可以限制协议访问的用户组。这不是个大问题，因为这些接口被作业令牌（参见 5.2.3 节）进一步保护着，访问接口前必须提交作业令牌。

大多数协议可以分为两类：需要被客户端访问的协议；管理协议。如果想限制只有白名单里的用户或组可以使用 Hadoop，可以将客户端协议设为更严格的值。但是注意，`security.job.task.protocol.acl` (YARN/MR2) 和 `security.task.umbilical.protocol.acl` (MR1) 必须始终设为 `*`。这么要求是因为，使用这些协议的用户通常被设为 MapReduce 作业的作业 ID，每个作业 ID 都会变，而且也不会出现在为你的集群配置的用户组里。因此，如果这些属性设置为 `*` 之外的其他值，可能导致作业失败。下面看两个用户会话，首先使用 `hadoop-policy.xml` 中的默认配置（示例 6-2），然后使用表格中的推荐值（示例 6-3）。

示例 **6-2**　使用默认的服务级授权策略

```
[alice@hadoop01 ~]$ hdfs dfs -ls .
Found 2 items
drwx------   - alice alice          0 2014-03-29 18:59 .Trash
drwx------   - alice alice          0 2014-03-29 18:59 .staging

[alice@hadoop01 ~]$ hdfs dfs -put file.txt .

[alice@hadoop01 ~]$ hdfs dfs -rm file.txt
14/03/29 21:26:07 INFO fs.TrashPolicyDefault: Namenode trash configuration
  Deletion interval = 1440 minutes, Emptier interval = 0 minutes.
Moved: 'hdfs://hadoop02:8020/user/alice/file.txt' to trash at:
  hdfs://hadoop02:8020/user/alice/.Trash/Current

[alice@hadoop01 ~]$ hdfs dfs -expunge
14/03/29 21:26:08 INFO fs.TrashPolicyDefault: Namenode trash configuration
  Deletion interval = 1 minutes, Emptier interval = 0 minutes.
14/03/29 21:26:09 INFO fs.TrashPolicyDefault: Deleted trash checkpoint:
  /user/alice/.Trash/140329185911
```

```
14/03/29 21:26:09 INFO fs.TrashPolicyDefault: Created trash checkpoint:
  /user/alice/.Trash/140329212609

[alice@hadoop01 ~]$ hdfs groups
alice@CLOUDERA : alice production-etl hadoop-users

[alice@hadoop01 ~]$ hdfs dfsadmin -refreshNodes
refreshNodes: Access denied for user alice. Superuser privilege is require

[alice@hadoop01 ~]$ hdfs dfsadmin -refreshServiceAcl

[alice@hadoop01 ~]$ hdfs dfsadmin -refreshUserToGroupsMappings

[alice@hadoop01 ~]$ hdfs dfsadmin -refreshSuperUserGroupsConfiguration

[alice@hadoop01 ~]$ yarn rmadmin -refreshQueues

14/03/29 21:26:16 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033

[alice@hadoop01 ~]$ yarn rmadmin -refreshNodes
14/03/29 21:26:18 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033

[alice@hadoop01 ~]$ yarn rmadmin -refreshSuperUserGroupsConfiguration
14/03/29 21:26:19 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033

[alice@hadoop01 ~]$ yarn rmadmin -refreshUserToGroupsMappings
14/03/29 21:26:21 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033

[alice@hadoop01 ~]$ yarn rmadmin -refreshAdminAcls
14/03/29 21:26:22 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033

[alice@hadoop01 ~]$ yarn rmadmin -refreshServiceAcl
14/03/29 21:26:23 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033

[alice@hadoop01 ~]$ yarn rmadmin -getGroups alice
14/03/29 21:26:25 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033
alice : alice production-etl hadoop-users

[alice@hadoop01 ~]$ yarn jar /opt/cloudera/parcels/CDH/lib/
  hadoop-mapreduce/hadoop-mapreduce-examples.jar randomtextwriter random-t
14/03/29 21:26:26 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8032
Running 30 maps.
Job started: Sat Mar 29 21:26:27 EDT 2014
14/03/29 21:26:27 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8032
```

```
14/03/29 21:26:27 INFO hdfs.DFSClient: Created HDFS_DELEGATION_TOKEN
  token 10 for alice on 172.25.2.223:8020
14/03/29 21:26:27 INFO security.TokenCache: Got dt for hdfs://hadoop02:802
  Kind: HDFS_DELEGATION_TOKEN, Service: 172.25.2.223:8020, Ident:
  (HDFS_DELEGATION_TOKEN token 10 for alice)
14/03/29 21:26:28 INFO mapreduce.JobSubmitter: number of splits:30
14/03/29 21:26:28 INFO mapreduce.JobSubmitter: Submitting tokens for job:
  job_1396142628007_0001
14/03/29 21:26:28 INFO mapreduce.JobSubmitter: Kind: HDFS_DELEGATION_TOKEN
  Service: 172.25.2.223:8020, Ident: (HDFS_DELEGATION_TOKEN token 10 for a
14/03/29 21:26:29 INFO impl.YarnClientImpl: Submitted application
  application_1396142628007_0001
14/03/29 21:26:29 INFO mapreduce.Job: The url to track the job:
  http://hadoop02:8088/proxy/application_1396142628007_0001/
14/03/29 21:26:29 INFO mapreduce.Job: Running job: job_1396142628007_0001
14/03/29 21:26:38 INFO mapreduce.Job: Job job_1396142628007_0001 running
  in uber mode : false
14/03/29 21:26:38 INFO mapreduce.Job:  map 0% reduce 0%
14/03/29 21:28:37 INFO mapreduce.Job:  map 3% reduce 0%
14/03/29 21:28:47 INFO mapreduce.Job:  map 7% reduce 0%
14/03/29 21:28:53 INFO mapreduce.Job:  map 10% reduce 0%
14/03/29 21:29:09 INFO mapreduce.Job:  map 17% reduce 0%
14/03/29 21:29:16 INFO mapreduce.Job:  map 23% reduce 0%
14/03/29 21:29:17 INFO mapreduce.Job:  map 27% reduce 0%
14/03/29 21:29:18 INFO mapreduce.Job:  map 30% reduce 0%
14/03/29 21:29:19 INFO mapreduce.Job:  map 33% reduce 0%
14/03/29 21:29:22 INFO mapreduce.Job:  map 50% reduce 0%
14/03/29 21:29:23 INFO mapreduce.Job:  map 60% reduce 0%
14/03/29 21:29:25 INFO mapreduce.Job:  map 70% reduce 0%
14/03/29 21:29:31 INFO mapreduce.Job:  map 77% reduce 0%
14/03/29 21:30:05 INFO mapreduce.Job:  map 83% reduce 0%
14/03/29 21:30:10 INFO mapreduce.Job:  map 90% reduce 0%
14/03/29 21:30:12 INFO mapreduce.Job:  map 93% reduce 0%
14/03/29 21:30:14 INFO mapreduce.Job:  map 97% reduce 0%
14/03/29 21:30:15 INFO mapreduce.Job:  map 100% reduce 0%
14/03/29 21:30:15 INFO mapreduce.Job: Job job_1396142628007_0001
  completed successfully
14/03/29 21:30:15 INFO mapreduce.Job: Counters: 29
        File System Counters
                FILE: Number of bytes read=0
                FILE: Number of bytes written=2679890
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=4550
                HDFS: Number of bytes written=33067041057
                HDFS: Number of read operations=120
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=60
        Job Counters
                Launched map tasks=30
                Other local map tasks=30
                Total time spent by all maps in occupied slots (ms)=401533
```

```
                Total time spent by all reduces in occupied slots (ms)=0
        Map-Reduce Framework
                Map input records=30
                Map output records=49159093
                Input split bytes=4550
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=22565
                CPU time spent (ms)=808110
                Physical memory (bytes) snapshot=12234526720
                Virtual memory (bytes) snapshot=40489713664
                Total committed heap usage (bytes)=12699172864
        org.apache.hadoop.examples.RandomTextWriter$Counters
                BYTES_WRITTEN=32212265105
                RECORDS_WRITTEN=49159093
        File Input Format Counters
                Bytes Read=0
        File Output Format Counters
                Bytes Written=33067041057
Job ended: Sat Mar 29 21:30:15 EDT 2014
The job took 227 seconds.

[alice@hadoop01 ~]$ hdfs dfs -rm -r random-text
14/03/29 21:30:17 INFO fs.TrashPolicyDefault: Namenode trash configuration
  Deletion interval = 1440 minutes, Emptier interval = 0 minutes.
Moved: 'hdfs://hadoop02:8020/user/alice/random-text' to trash at:
  hdfs://hadoop02:8020/user/alice/.Trash/Current

[alice@hadoop01 ~]$ hdfs dfs -expunge
14/03/29 21:30:18 INFO fs.TrashPolicyDefault: Namenode trash configuration
  Deletion interval = 1 minutes, Emptier interval = 0 minutes.
14/03/29 21:30:19 INFO fs.TrashPolicyDefault: Deleted trash checkpoint:
  /user/alice/.Trash/140329212609
14/03/29 21:30:19 INFO fs.TrashPolicyDefault: Created trash checkpoint:
  /user/alice/.Trash/140329213019
```

示例 6-2 展示了 Alice 在使用一系列用户和管理命令。使用默认的服务级授权策略时，虽然有一些命令（如 `hdfs dfsadmin -refreshNodes`）需要超级用户权限，但很多命令使用默认的服务级授权策略时，并不需要特殊权限。示例 6-3 使用前面介绍的推荐策略执行了完全相同的一系列命令。

示例 **6-3**　使用推荐的服务级授权策略

```
[alice@hadoop01 ~]$ hdfs dfs -ls .
Found 2 items
drwx------   - alice alice          0 2014-03-29 18:52 .Trash
drwx------   - alice alice          0 2014-03-29 18:45 .staging
```

```
[alice@hadoop01 ~]$ hdfs dfs -put file.txt .

[alice@hadoop01 ~]$ hdfs dfs -rm file.txt
14/03/29 18:54:11 INFO fs.TrashPolicyDefault: Namenode trash configuration
  Deletion interval = 1440 minutes, Emptier interval = 0 minutes.
Moved: 'hdfs://hadoop02:8020/user/alice/file.txt' to trash at:
  hdfs://hadoop02:8020/user/alice/.Trash/Current

[alice@hadoop01 ~]$ hdfs dfs -expunge
14/03/29 18:54:13 INFO fs.TrashPolicyDefault: Namenode trash configuration
  Deletion interval = 1 minutes, Emptier interval = 0 minutes.
14/03/29 18:54:13 INFO fs.TrashPolicyDefault: Deleted trash checkpoint:
  /user/alice/.Trash/140329185237
14/03/29 18:54:13 INFO fs.TrashPolicyDefault: Created trash checkpoint:
  /user/alice/.Trash/140329185413

[alice@hadoop01 ~]$ hdfs groups
alice@CLOUDERA : alice production-etl hadoop-users

[alice@hadoop01 ~]$ hdfs dfsadmin -refreshNodes
refreshNodes: Access denied for user alice. Superuser privilege is require

[alice@hadoop01 ~]$ hdfs dfsadmin -refreshServiceAcl
refreshServiceAcl: User alice@CLOUDERA (auth:KERBEROS) is not authorized f
  protocol interface
  org.apache.hadoop.security.authorize.RefreshAuthorizationPolicyProtocol,
  expected client Kerberos principal is null

[alice@hadoop01 ~]$ hdfs dfsadmin -refreshUserToGroupsMappings
refreshUserToGroupsMappings: User alice@CLOUDERA (auth:KERBEROS) is not
  authorized for protocol interface
  org.apache.hadoop.security.RefreshUserMappingsProtocol, expected client
  Kerberos principal is null

[alice@hadoop01 ~]$ hdfs dfsadmin -refreshSuperUserGroupsConfiguration
refreshSuperUserGroupsConfiguration: User alice@CLOUDERA (auth:KERBEROS) i
  not authorized for protocol interface
  org.apache.hadoop.security.RefreshUserMappingsProtocol, expected client
  Kerberos principal is null

[alice@hadoop01 ~]$ yarn rmadmin -refreshQueues
14/03/29 18:54:21 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033
refreshQueues: User alice@CLOUDERA (auth:KERBEROS) is not authorized
  for protocol interface
  org.apache.hadoop.yarn.server.api.ResourceManagerAdministrationProtocolP
  expected client Kerberos principal is null

[alice@hadoop01 ~]$ yarn rmadmin -refreshNodes
14/03/29 18:54:22 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033
refreshNodes: User alice@CLOUDERA (auth:KERBEROS) is not authorized
```

```
  for protocol interface
  org.apache.hadoop.yarn.server.api.ResourceManagerAdministrationProtocolP
  expected client Kerberos principal is null

[alice@hadoop01 ~]$ yarn rmadmin -refreshSuperUserGroupsConfiguration
14/03/29 18:54:24 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033
refreshSuperUserGroupsConfiguration: User alice@CLOUDERA (auth:KERBEROS)
  is not authorized for protocol interface
  org.apache.hadoop.yarn.server.api.ResourceManagerAdministrationProtocolP
  expected client Kerberos principal is null

[alice@hadoop01 ~]$ yarn rmadmin -refreshUserToGroupsMappings
14/03/29 18:54:25 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033
refreshUserToGroupsMappings: User alice@CLOUDERA (auth:KERBEROS)
  is not authorized for protocol interface
  org.apache.hadoop.yarn.server.api.ResourceManagerAdministrationProtocolP
  expected client Kerberos principal is null

[alice@hadoop01 ~]$ yarn rmadmin -refreshAdminAcls
14/03/29 18:54:26 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033
refreshAdminAcls: User alice@CLOUDERA (auth:KERBEROS)
  is not authorized for protocol interface
  org.apache.hadoop.yarn.server.api.ResourceManagerAdministrationProtocolP
  expected client Kerberos principal is null

[alice@hadoop01 ~]$ yarn rmadmin -refreshServiceAcl
14/03/29 18:54:28 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033
refreshServiceAcl: User alice@CLOUDERA (auth:KERBEROS)
  is not authorized for protocol interface
  org.apache.hadoop.yarn.server.api.ResourceManagerAdministrationProtocolP
  expected client Kerberos principal is null

[alice@hadoop01 ~]$ yarn rmadmin -getGroups alice
14/03/29 18:54:29 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8033
getGroups: User alice@CLOUDERA (auth:KERBEROS)
  is not authorized for protocol interface
  org.apache.hadoop.yarn.server.api.ResourceManagerAdministrationProtocolP
  expected client Kerberos principal is null

[alice@hadoop01 ~]$ yarn jar /opt/cloudera/parcels/CDH/lib/hadoop-mapreduc
  hadoop-mapreduce-examples.jar randomtextwriter random-text
14/03/29 18:54:31 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8032
Running 30 maps.
Job started: Sat Mar 29 18:54:32 EDT 2014
14/03/29 18:54:32 INFO client.RMProxy: Connecting to ResourceManager at
  hadoop02/172.25.2.223:8032
14/03/29 18:54:32 INFO hdfs.DFSClient: Created HDFS_DELEGATION_TOKEN
```

```
  token 9 for alice on 172.25.2.223:8020
14/03/29 18:54:32 INFO security.TokenCache: Got dt for hdfs://hadoop02:802
  Kind: HDFS_DELEGATION_TOKEN, Service: 172.25.2.223:8020, Ident:
  (HDFS_DELEGATION_TOKEN token 9 for alice)
14/03/29 18:54:32 INFO mapreduce.JobSubmitter: number of splits:30
14/03/29 18:54:32 INFO mapreduce.JobSubmitter: Submitting tokens for job:
  job_1396131817617_0003
14/03/29 18:54:32 INFO mapreduce.JobSubmitter: Kind: HDFS_DELEGATION_TOKEN
  Service: 172.25.2.223:8020, Ident: (HDFS_DELEGATION_TOKEN token 9 for al
14/03/29 18:54:33 INFO impl.YarnClientImpl: Submitted application
  application_1396131817617_0003
14/03/29 18:54:33 INFO mapreduce.Job: The url to track the job:
  http://hadoop02:8088/proxy/application_1396131817617_0003/
14/03/29 18:54:33 INFO mapreduce.Job: Running job: job_1396131817617_0003
14/03/29 18:54:40 INFO mapreduce.Job: Job job_1396131817617_0003
  running in uber mode : false
14/03/29 18:54:40 INFO mapreduce.Job:  map 0% reduce 0%
14/03/29 18:56:20 INFO mapreduce.Job:  map 3% reduce 0%
14/03/29 18:56:53 INFO mapreduce.Job:  map 7% reduce 0%
14/03/29 18:56:57 INFO mapreduce.Job:  map 10% reduce 0%
14/03/29 18:56:59 INFO mapreduce.Job:  map 13% reduce 0%
14/03/29 18:57:02 INFO mapreduce.Job:  map 17% reduce 0%
14/03/29 18:57:15 INFO mapreduce.Job:  map 20% reduce 0%
14/03/29 18:57:36 INFO mapreduce.Job:  map 27% reduce 0%
14/03/29 18:57:44 INFO mapreduce.Job:  map 30% reduce 0%
14/03/29 18:57:59 INFO mapreduce.Job:  map 33% reduce 0%
14/03/29 18:58:09 INFO mapreduce.Job:  map 37% reduce 0%
14/03/29 18:58:19 INFO mapreduce.Job:  map 40% reduce 0%
14/03/29 18:58:23 INFO mapreduce.Job:  map 43% reduce 0%
14/03/29 18:58:25 INFO mapreduce.Job:  map 47% reduce 0%
14/03/29 18:58:35 INFO mapreduce.Job:  map 50% reduce 0%
14/03/29 18:58:36 INFO mapreduce.Job:  map 53% reduce 0%
14/03/29 18:58:39 INFO mapreduce.Job:  map 57% reduce 0%
14/03/29 18:58:40 INFO mapreduce.Job:  map 60% reduce 0%
14/03/29 18:58:44 INFO mapreduce.Job:  map 63% reduce 0%
14/03/29 18:58:45 INFO mapreduce.Job:  map 67% reduce 0%
14/03/29 18:58:47 INFO mapreduce.Job:  map 70% reduce 0%
14/03/29 18:58:53 INFO mapreduce.Job:  map 73% reduce 0%
14/03/29 18:58:55 INFO mapreduce.Job:  map 80% reduce 0%
14/03/29 18:58:57 INFO mapreduce.Job:  map 83% reduce 0%
14/03/29 18:59:01 INFO mapreduce.Job:  map 90% reduce 0%
14/03/29 18:59:05 INFO mapreduce.Job:  map 93% reduce 0%
14/03/29 18:59:07 INFO mapreduce.Job:  map 100% reduce 0%
14/03/29 18:59:07 INFO mapreduce.Job: Job job_1396131817617_0003
  completed successfully
14/03/29 18:59:07 INFO mapreduce.Job: Counters: 29
        File System Counters
                FILE: Number of bytes read=0
                FILE: Number of bytes written=2679890
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=4550
```

```
                HDFS: Number of bytes written=33067034387
                HDFS: Number of read operations=120
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=60
        Job Counters
                Launched map tasks=30
                Other local map tasks=30
                Total time spent by all maps in occupied slots (ms)=53191
                Total time spent by all reduces in occupied slots (ms)=0
        Map-Reduce Framework
                Map input records=30
                Map output records=49157281
                Input split bytes=4550
                Spilled Records=0
                Failed Shuffles=0
                Merged Map outputs=0
                GC time elapsed (ms)=13711
                CPU time spent (ms)=741910
                Physical memory (bytes) snapshot=10065694720
                Virtual memory (bytes) snapshot=40491339776
                Total committed heap usage (bytes)=14946533376
        org.apache.hadoop.examples.RandomTextWriter$Counters
                BYTES_WRITTEN=32212267432
                RECORDS_WRITTEN=49157281
        File Input Format Counters
                Bytes Read=0
        File Output Format Counters
                Bytes Written=33067034387
Job ended: Sat Mar 29 18:59:07 EDT 2014
The job took 275 seconds.

[alice@hadoop01 ~]$ hdfs dfs -rm -r random-text
14/03/29 18:59:09 INFO fs.TrashPolicyDefault: Namenode trash
  configuration: Deletion interval = 1440 minutes, Emptier
  interval = 0 minutes.
Moved: 'hdfs://hadoop02:8020/user/alice/random-text' to trash
  at: hdfs://hadoop02:8020/user/alice/.Trash/Current

[alice@hadoop01 ~]$ hdfs dfs -expunge
14/03/29 18:59:10 INFO fs.TrashPolicyDefault: Namenode trash
  configuration: Deletion interval = 1 minutes, Emptier
  interval = 0 minutes.
14/03/29 18:59:11 INFO fs.TrashPolicyDefault: Deleted trash checkpoint:
  /user/alice/.Trash/140329185413
14/03/29 18:59:11 INFO fs.TrashPolicyDefault: Created trash checkpoint:
  /user/alice/.Trash/140329185911
```

这次 Alice 仍然能够访问所有用户功能，但管理员功能被禁止了，报错信息为 User alice@CLOUDERA (auth:KERBEROS) is not authorized for protocol interface < 协议名 >。这表示该用户既不在该协议的 ACL 表里，也不属于 ACL 表中的某个组。服

务级授权虽然复杂，却是一种控制 Hadoop 集群访问的有力工具。例如，在我们所配的策略控制下，hdfs 用户不再具有查看或修改 HDFS 中文件的权限，除非它被添加到 hadoop-users 组。这对于需要由某个管理动作追溯执行该动作的管理员的公司来说，非常有用。如果将 dfs.permissions.supergroup 设置为 hadoop-admins，以结合服务级授权，就能将管理动作绑定到特定账号。示例 6-4 展示了 hdfs 用户尝试列出 Alice 主目录下的文件以及删除她所上传的一个文件时，将会如何。

示例 **6-4**　用户 hdfs 被拒绝访问 ClientProtocol

```
[hdfs@hadoop01 ~]$ hdfs dfs -ls /user/alice/
ls: User hdfs@CLOUDERA (auth:KERBEROS) is not authorized for protocol inte
  org.apache.hadoop.hdfs.protocol.ClientProtocol, expected client Kerberos
is null
[hdfs@hadoop01 ~]$ hdfs dfs -rm /user/alice/file.txt
rm: User hdfs@CLOUDERA (auth:KERBEROS) is not authorized for protocol inte
  org.apache.hadoop.hdfs.protocol.ClientProtocol, expected client Kerberos
is null
```

注意，用户 hdfs 在协议层就被拒绝访问了，这发生在 HDFS 层执行的权限检查之前。虽然从文件系统的角度看，hdfs 是超级用户，但根据此前的服务级检查，它不能查看和修改任何数据。示例 6-5 展示了一个 hadoop-admins 组成员 Joey 尝试执行相同的动作时，情况会如何。

示例 **6-5**　hadoop-admins 组的一个成员删除用户文件

```
[joey@hadoop01 ~]$ hdfs dfs -ls /user/alice/
Found 3 items
drwx------   - alice alice          0 2014-03-29 21:30 /user/alice/.Trash
drwx------   - alice alice          0 2014-03-29 21:30 /user/alice/.stagin
-rw-------   3 alice alice          5 2014-03-29 21:48 /user/alice/file.tx
[joey@hadoop01 ~]$ hdfs dfs -rm /user/alice/file.txt
14/03/29 21:49:26 INFO fs.TrashPolicyDefault: Namenode trash configuration
  interval = 1440 minutes, Emptier interval = 0 minutes.
Moved: 'hdfs://hadoop02:8020/user/alice/file.txt' to trash at:
  hdfs://hadoop02:8020/user/joey/.Trash/Current
[joey@hadoop01 ~]$ hdfs groups joey
joey : joey hadoop-admins hadoop-users
[joey@hadoop01 ~]$ hdfs groups hdfs
hdfs : hdfs hadoop
```

你会注意到，这次这些行为被允许了。因为 Joey 同时是 `hadoop-admins` 组（被配置为 HDFS 中的超级用户组）和 `hadoop-users` 组（使用户能够访问 HDFS 客户端协议）的成员。

除了在 `hadoop-policy.xml` 配置 ACL 之外，某些 HDFS 管理动作（如强制执行 HA 故障转移）也只对 HDFS 集群管理员可用。管理员可以使用 `hdfs-site.xml` 中的 `dfs.cluster.administrators` 进行配置，其值为以逗号分隔的、可以管理 HDFS 的用户列表和组列表，两个列表之间用空格分隔。开头的空格表示空的用户列表，结尾的空格表示空的组列表。特殊值 `*` 表示所有用户具有对 HDFS 的管理权限，值 " "（不包含引号）表示所有用户都没有管理权限（这是默认设置）。示例 6-6 给出了我们的示例环境下推荐的配置。

示例 **6-6**　hdfs-site.xml 中的 dfs.cluster.administrators 配置

```
<property>
  <name>dfs.cluster.administrators</name>
  <value>hdfs hadoop-admins</value>
</property>
```

管理 MapReduce Job History Server 的 ACL 不在 `hadoop-policy.xml` 文件里配置，而是要配置 `mapred-site.xml` 中的 `mapreduce.jobhistory.admin.acl`，其值为逗号分隔的用户列表和组列表，格式与本节开始部分以及表 6-1 中描述的一致。特殊值 `*` 表示允许所有用户管理 JobHistory Server（这是默认配置）。示例 6-7 给出了推荐的配置。

示例 **6-7**　mapred-site.xml 中的 mapreduce.jobhistory.admin.acl 配置

```
<property>
  <name>mapreduce.jobhistory.admin.acl</name>
  <value>mapred hadoop-admins</value>
</property>
```

6.3　MapReduce和YARN的授权

MapReduce 和 YARN 都不控制对数据的访问，但都提供了对集群资

源（如 CPU、内存、硬盘 I/O 和网络 I/O）的访问。由于这些资源是有限的，因此管理员通常要为特定用户或组分配资源，尤其在多用户环境中更是如此。上一节描述的服务级授权控制了对特定协议的访问，比如谁可以向集群提交作业而谁不可以，但对于控制集群资源的访问还不够细化。MapReduce（MR1）和 YARN 都支持以作业队列的方式限制如何给作业分配资源。为了安全地控制这些资源，Hadoop 支持作业队列上的访问控制列表（ACL）。这些 ACL 控制了哪些用户可以提交到特定队列，以及哪些用户可以管理队列。MapReduce 定义了不同类型的用户，这影响 ACL 的解释方式。

MapReduce/YARN 集群所有者

启动 JobTracker 进程（MR1）或 ResourceManager 进程（YARN）的用户被定义为集群所有者。该用户拥有提交作业到任意队列的权限，并能管理任意队列或作业。大多数情况下，集群所有者在 MapReduce（MR1）中即是 `mapred`，在 YARN 中即是 `yarn`。由于以集群所有者的身份运行作业很危险，因此 LinuxTaskController 默认把 `mapred` 和 `yarn` 用户列入黑名单，使其无法提交作业。

MapReduce 管理员

可以通过设置以创建与集群所有者权限相同的全局 MapReduce 管理员。定义特定的用户或组为管理员的好处是，可以对每个管理员各自的动作进行审计。这也能避免不得不将密码分配到共享账号而导致被破解机率增大的情况。

作业所有者

作业所有者是提交该作业的用户。作业所有者总是可以管理他们自己的作业，但只有被授予作业提交权限的用户才能提交作业到队列。

队列管理员

用户或组可以被授予对队列中所有作业的管理权限，队列管理员也可以提交作业到他所管理的队列。

6.3.1　MapReduce（MR1）

对于 MR1，ACL 是全局管理的，并应用于任意支持 ACL 的作业调度器。CapacityScheduler 和 FairScheduler 都支持 ACL，但 FIFO（默

认的）调度器不支持。配置单队列 ACL 前，需要启用 MapReduce ACL，配置 MapReduce 管理员，以及在 `mapred-site.xml` 中定义队列名。

`mapred.acls.enabled`

设为 `true` 时，提交或管理作业后会检查 ACL。在 JobTracker 接口中对查看和修改作业进行授权时，也要检查 ACL。

`mapreduce.cluster.administrators`

为 MapReduce 集群配置管理员。无论作业或队列相关的 ACL 如何配置，集群管理员总可以管理任意作业或队列。该配置的格式是一个逗号分隔的、可以访问该协议的用户列表和组列表。两个列表之间使用空格分隔，开头的空格表示空的用户列表，结尾的空格表示空的组列表。特殊值 `*` 表示允许所有用户访问该协议（这是默认配置）。具体示例见表 6-1。

`mapred.queue.names`

一系列以逗号分隔的队列名。如果要为某个队列配置 ACL，则该队列必须被列在该属性里。MapReduce 通常支持至少一个名为 `default` 的队列，因此该参数应当始终在定义的队列列表中包含 `default`。

单队列 ACL 配置存储在 `mapred-queue-acls.xml` 里。每个队列有两种 ACL 可以配置，分别是提交 ACL 和管理 ACL。

`mapred.queue.<queue_name>.acl-submit-job`

可以向队列 `queue_name` 提交作业的用户的访问控制表。提交作业的 ACL 的格式为逗号分隔的、允许向该队列提交作业的用户列表和组列表，其格式与 6.2 节和表 6-1 中描述的一致。特殊值 `*` 表示允许所有用户访问该协议（这是默认设置）。但无论该项如何设置，集群所有者和 MapReduce 管理员都可以提交作业。

`mapred.queue.<queue_name>.acl-administer-jobs`

可以对队列 `queue_name` 中的所有作业进行查看详情、终止或修改优先级操作的用户的访问控制表。管理作业的 ACL 的格式为逗号分隔的、允许管理该队列中作业的用户列表和组列表，其格式与 6.2 节和表 6-1 中描述的一致。特殊值 `*` 表示允许所有用户访问该协

议（这是默认设置）。但无论该项如何设置，集群所有者和 MapReduce 管理员都可以管理所有队列中的所有作业。作业所有者也可以管理作业。

除单队列 ACL 外，还有两种可以为每个作业配置的 ACL。这些设置的默认值可以放在客户端使用的 `mapred- site.xml` 文件里，这些值也可以被某个作业重写。

`mapreduce.job.acl-view-job`

允许查看作业详情的用户的访问控制表。查看作业的 ACL 的格式为逗号分隔的、允许查看作业详情的用户列表和组列表，其格式与 6.2 节和表 6-1 中描述的一致。特殊值 `*` 表示允许所有用户访问该协议（这是默认设置）。但无论该项如何设置，集群所有者、MapReduce 管理员和作业提交到的队列的管理员都可以查看作业。该 ACL 控制了对作业级计数器、任务级计数器、任务的诊断信息、TaskTracker Web 界面展示的任务日志以及 JobTracker Web 界面展示的 `job.xml` 的访问。

`mapreduce.job.acl-modify-job`

允许停止作业、结束任务、放弃任务、设置作业优先级的用户的访问控制表。修改作业的 ACL 的格式为逗号分隔的、允许修改作业的用户列表和组列表，其格式与 6.2 节和表 6-1 中描述的一致。特殊值 `*` 表示允许所有用户访问该协议（这是默认设置）。但无论该项如何设置，作业所有者、集群所有者、MapReduce 管理员和作业提交到的队列的管理员都可以修改作业。

如果有这么一个部署场景，你想为访问作业详情配置一个默认拒绝的策略，那么对于这两种配置而言，一个明智的默认值就是一个空格 `" "`（不包含引号）。这会禁止除作业所有者、队列管理员、集群管理员和集群所有者之外的所有用户访问作业细节。

> 为了控制对 JobTracker Web 界面中作业详情的访问，必须如前所述配置 MapReduce ACL，也需要按照第 11 章描述的方法启用 Web 界面的授权。

6.3.2　YARN (MR2)

YARN/MR2 中，队列 ACL 不再是全局定义的，每个调度器都提供了自己定义 ACL 的方法。ACL 仍然是全局启用的，有一个全局的 ACL

定义 YARN 管理员。启用 YARN ACL 以及定义管理员的选项在 `yarn-site.xml` 里进行配置。示例 6-8 提供了一些配置示例。

示例 **6-8**　yarn-site.xml 中的 YARN ACL 配置

```
<property>
  <name>yarn.acl.enable</name>
  <value>true</value>
</property>
<property>
  <name>yarn.admin.acl</name>
  <value>yarn hadoop-admins</value>
</property>
```

由于每个调度器的配置各不相同，我们将逐个梳理如何设置队列 ACL。对于这两种例子，我们会实现相同的用例。我们的集群主要用于运行生产 ETL 流水线和用于生成日常报告的生产查询，也有一些临时报告，但生产作业应优先处理。为了进行访问控制，我们额外定义了两个用户组，这两个组只包含先前定义的 `hadoop-users` 的一部分。一个是 `production-etl` 组，它包含了运行生产 ETL 作业的用户；另一个是 `production-queries` 组，它包含了运行生产查询的用户。该例子中，Alice 是 `production-etl` 组的成员，Bob 是 `production-queries` 组的成员。先配置 FairScheduler。

01. **FairScheduler**

为了保证生产作业所需的资源，首先必须禁用 FairScheduler 的默认行为，即将每个用户放到与其用户名相匹配的队列。可以通过设置两个参数 `yarn.scheduler.fair.user-as-default_queue` 和 `yarn.scheduler.fair.allow-undeclared-pools` 的值为 `false` 实现。第一个参数将默认的队列改为 `default`，第二个参数确保用户无法向未预定义的队列提交作业。这些设置以及开启 FairScheduler 的设置项在示例 6-9 中都能找到。

示例 **6-9**　yarn-site.xml 中的 FairScheduler 配置

```
<property>
  <name>yarn.resourcemanager.scheduler.class</name>
  <value>
    org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.Fa
  </value>
```

```
</property>
<property>
  <name>yarn.scheduler.fair.user-as-default-queue</name>
  <value>false</value>
</property>
<property>
  <name>yarn.scheduler.fair.allow-undeclared-pools</name>
  <value>false</value>
</property>
```

接下来，在 `fair-scheduler.xml` 文件中定义队列及其 ACL。FairScheduler 使用分级队列系统，每个队列都是 `root` 队列的子队列。示例中，我们想将集群资源中的 90% 分配给生产作业，10% 分配给临时作业。为了实现这点，我们定义了两个 root 队列的子队列：生产作业的是 `prod`，临时作业的是 `default`。之所以将临时队列称为 default，是因为如果没有为某个作业指定队列，那么该作业就会被提交到这里。资源管理是个复杂的话题，我们可以调节各种各样的设置，以井然有序地控制资源。由于我们关注的是安全方面，因此将用一个简化的方式使用队列权重控制资源。需要了解的是，所有队列具有相同的父队列，它们的资源配置被定义为其权重除以其所有兄弟队列的权重。该例子中，我们可以为 `prod` 分配权重 9.0，为 `default` 分配权重 1.0，这样就能得到 90 ： 10 的划分比例。

我们还想将生产队列分成两个子队列：一个用于 ETL 作业，一个用于请求。本例中，我们将把两个队列的权重都设为 1.0，因此两个队列的权重是一样的。值得注意的是，份额的计算发生在父队列上下文中。本例中，意思就是由于我们将 `etl` 和 `queries` 的队列都设成 `prod` 队列 50% 的资源，最终它们获得的资源是全局等额分量，每个队列是 45%（50% × 90% = 45%）。

资源是继承的，ACL 也是。使用 FairScheduler 后，能提交作业到某个队列的用户也能提交作业到其子队列，这也同样适用于队列的管理权限。和前面的例子一样，我们想使 `hadoop-admins` 组的任意成员都能管理任意作业 / 队列，就要将他们加到根队列的 `aclAdministerApps` ACL 中。值得注意的是，必须将 `aclSubmitApps` ACL 设成 " "（不含引号），否则，由于没有定义的默认 ACL 允许所有，因此会导致所有用户都能提交作业到任意队列。对于默认队列，我们想要允许 `hadoop-users` 组的任意成员都能提交作业，因此要将

aclSubmitApps 设成 " hadoop-users"（不含引号，并注意开头有空格）。对于 prod.etl 和 prod.queries 队列，则要将 aclSubmitApps 分别设为 " production-etl" 和 " production-queries"（不含引号），这些都是我们之前定义的组。完整的 FairScheduler 配置如示例 6-10 所示。

示例 **6-10** fair-scheduler.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<allocations>
    <queue name="root">
        <weight>1.0</weight>
        <aclSubmitApps> </aclSubmitApps>
        <aclAdministerApps> hadoop-admins</aclAdministerApps>
        <queue name="prod">
            <weight>9.0</weight>
            <aclSubmitApps> </aclSubmitApps>
            <aclAdministerApps> </aclAdministerApps>
            <queue name="etl">
                <weight>1.0</weight>
                <aclSubmitApps> production-etl</aclSubmitApps>
                <aclAdministerApps>alice </aclAdministerApps>
            </queue>
            <queue name="queries">
                <weight>1.0</weight>
                <aclSubmitApps> production-queries</aclSubmitApps>
                <aclAdministerApps>bob </aclAdministerApps>
            </queue>
        </queue>
        <queue name="default">
            <weight>1.0</weight>
            <aclSubmitApps> hadoop-users</aclSubmitApps>
            <aclAdministerApps> </aclAdministerApps>
        </queue>
    </queue>
</allocations>
```

现在看看 Bob 在没有预先定义队列 ACL 的情况下，尝试终止一个 Alice 的作业会发生什么。首先，Bob 得到运行着的作业列表，并找到 Alice 作业的 JobId，然后发送终止作业的请求。由于缺乏对谁能管理作业而谁不能管理作业的控制，YARN 会很乐意满足他的请求。示例 6-11 给出了 Bob 的完整用户会话。

示例 **6-11** 未定义 ACL 的情况下终止其他用

户的作业

```
[bob@hadoop01 ~]$ mapred job -list
Total jobs:1
                  JobId      State           StartTime      UserName
 job_1396200012809_0002    RUNNING       1396201153018         alice
[bob@hadoop01 ~]$ mapred job -kill job_1396200012809_0002
Killed job job_1396200012809_0002
```

这并不理想，因为用户可以干扰其他用户的生产作业。更重要的是，一个简单的复制 / 粘贴错误就有可能导致一个用户不小心终止另外一个用户的作业。如果我们在 FairScheduler 中配置了 ACL 之后再执行相同的操作，就会得到示例 6-12 所示的结果。

示例 **6-12**　Bob 被队列 ACL 拒绝给予管理权限

```
[bob@hadoop01 ~]$ mapred job -list
Total jobs:1
                  JobId      State           StartTime           Use
 job_1396192703139_0001    RUNNING       1396192707596
[bob@hadoop01 ~]$ mapred job -kill job_1396192703139_0001
...
Exception in thread "main" java.io.IOException: org.apache.hadoop.ya
nException: java.security.AccessControlException: User bob cannot pe
ODIFY_APP on application_1396192703139_0001
        at org.apache.hadoop.yarn.ipc.RPCUtil.getRemoteException(RPC
...
[bob@hadoop01 ~]$
```

很多时候，一些管理员必须要能终止另外一个用户的作业，因此我们在根队列上将管理权限设置到 hadoop-admins 组。因此，如果其中一个管理员 Joey 尝试终止一个作业，系统将进行以下处理，如示例 6-13 所示。

示例 **6-13**　成功终止一个 MapReduce 作业

```
[joey@hadoop01 ~]$ mapred job -list
Total jobs:1
                  JobId      State           StartTime       UserNar
 job_1396192703139_0002    RUNNING       1396193202565          alic
[joey@hadoop01 ~]$ mapred job -kill job_1396192703139_0002
Killed job job_1396192703139_0002
```

控制管理权限显然是有用的，但防止用户提交作业到错误队列也很重要。我们的例子中，由于 Alice 是 production-etl 组的成员，因此她具有提交作业到 prod.etl 队列的权限。但她不是 production-queries 组的成员，因此如果尝试提交作业到该处，则会被拒绝，如示例 6-14 所示。

示例 **6-14**　Alice 被拒绝向 prod.queries 队列提交作业

```
[alice@hadoop01 ~]$ yarn jar \
  /opt/cloudera/parcels/CDH/lib/hadoop-mapreduce/hadoop-mapreduce-ex
  randomtextwriter -Dmapreduce.job.queuename=prod.queries random-tex
...
Job started: Sun Mar 30 13:20:57 EDT 2014
14/03/30 13:20:59 ERROR security.UserGroupInformation: PriviledgedAc
as:alice@CLOUDERA (auth:KERBEROS) cause:java.io.IOException: Failed
User alice cannot submit applications to queue root.prod.queries
...
```

02. **CapacityScheduler**

CapacityScheduler 与 FairScheduler 一样支持分级队列，也支持与 FairScheduler 一样的单队列 ACL 和 ACL 继承策略。实际上，从安全角度看，这两个调度器是相同的，唯一的区别在于配置文件的格式。为了实现与前所述相同的策略，需要先在 yarn-site.xml 中启用 CapacityScheduler，如示例 6-15 所示。

示例 **6-15**　yarn-site.xml 中的 CapacityScheduler 配置

```
<property>
 <name>
  yarn.resourcemanager.scheduler.class
 </name>
 <value>
  org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity.C
</value>
</property>
```

一旦启用，CapacityScheduler 就会从 capacity-scheduler.xml 文件读取配置。示例 6-16 展示了实现同一队列和 ACL 的配置。对于 FairScheduler，ACL 被配置为队列

定义的一个子元素，用 aclSubmitApps 标签控制谁可以提交应用到队列，用 aclAdministerApps 标签控制谁可以管理队列中的作业。对于 CapacityScheduler 来说，对应的设置分别是如下所示的属性（将 <path-to-queue > 替换为队列的层次结构）。

- yarn.scheduler.capacity.root.< *path-to-queue* >.acl_submit_applications
- yarn.scheduler.capacity.root.< *path-to-queue* >.acl_administer_applications

例如，定义 prod.etl 队列的 ACL 是 yarn.scheduler.capacity.root.prod.etl.acl_submit_ap
如示例 6-16 所示。

示例 **6-16**　capacity-scheduler.xml

```
<!-- Define ACLs and subqueues for the root queue -->
<property>
  <name>yarn.scheduler.capacity.root.acl_submit_applications</name>
  <value> </value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.acl_administer_applications</na
  <value> hadoop-admins</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.queues</name>
  <value>prod,default</value>
</property>

<!-- Define capacity and ACLs for the root.default queue -->
<property>
  <name>yarn.scheduler.capacity.root.default.capacity</name>
  <value>10.0</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.acl_submit_applications</name>
  <value> hadoop-users</value>
</property>

<!-- Define capacity, ACLs, and subqueues for the root.prod queue --
<property>
  <name>yarn.scheduler.capacity.root.prod.capacity</name>
  <value>90.0</value>
</property>
<property>
```

```
  <name>yarn.scheduler.capacity.root.prod.queues</name>
  <value>etl,queries</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.prod.acl_submit_applications</
  <value> </value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.prod.acl_administer_applicatio
  <value> </value>
</property>

<!-- Define capacity and ACLs for the root.prod.etl queue -->
<property>
  <name>yarn.scheduler.capacity.root.prod.etl.capacity</name>
  <value>50.0</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.prod.etl.acl_submit_applicatio
  <value> production-etl</value>
</property>
<property>
  <name>yarn.scheduler.capacity.root.prod.etl.acl_administer_applica
  <value>alice </value>
</property>

<!-- Define capacity and ACLs for the root.prod.queries queue -->
<property>
  <name>yarn.scheduler.capacity.root.prod.queries.capacity</name>
  <value>50.0</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.prod.queries.acl_submit_applicat
 <value> production-queries</value>
</property>
<property>
 <name>yarn.scheduler.capacity.root.prod.queries.acl_administer_appl
 <value>bob </value>
</property>
```

6.4 ZooKeeper ACLs

Apache ZooKeeper 使用 ACL 控制对 ZNode（路径）的访问。ZooKeeper 的 ACL 与 POSIX 权限位类似，但更加灵活，因为其权限设置在单用户上，而不是所有者和主要组上。实际上，ZooKeeper 里没有所有者或组的概念。5.2.2 节讲过，用户由一个身份验证模式和模式对应的 ID 确定，ID 的格式根据模式的不同而不同。

每个 ACL 包含一个模式、ID 和权限。可用的权限列表见表 6-5。值得注意的是，ZooKeeper 中，权限是非递归的，只作用于所属的 ZNode，不作用于其子节点。由于 ZooKeeper 中没有 ZNode 所有者的概念，因此用户需要有 ZNode 的 `ADMIN` 权限才能设置 ACL。

表**6-5**：**ZooKeeper ACL**权限

权限	描述
CREATE	创建子 Znode 的权限
READ	读取 ZNode 数据以及列出 ZNode 子节点的权限
WRITE	设置 ZNode 数据的权限
DELETE	删除子 Znode 的权限
ADMIN	设置 ACL 的权限

`CREATE` 和 `DELETE` 权限用于控制谁可以创建 ZNode 的子节点。授予 `CREATE` 权限但不授予 `DELETE` 权限的使用情况是：你希望用户可以在某个路径创建子节点，但该路径下只有管理员才能删除子节点。

如果使用 JAVA API 添加 ACL，首先要使用模式项和 ID 号创建一个 `ID` 对象，然后使用该 `ID` 以及整型的权限创建一个 `ACL` 对象。你可以手动计算权限的整型值，或者使用 `ZooDefs.Perms` 类中的常量得出想要设置的权限组合而成的整型值。如示例 6-17 所示，使用 Java 代码在某个路径上设置 ACL。

示例 **6-17**　使用 JAVA API 设置 ZooKeeper ACL

```
// 连接到ZooKeeper
ZooKeeper zk = new ZooKeeper("zk.example.com:2181", 60000, watcher);

// 使用密码secret为alice创建Id
Id id = new Id("digest", "alice:secret");

// 创建授予Alice READ和CREATE权限的ACL
ACL acl = new ACL(ZooDefs.Perms.READ | ZooDefs.Perms.CREATE, id);

zk.setAcl("/test", Arrays.asList(new ACL[] {acl}), -1);
```

5.2.2 节已经描述过 `digest` 模式，但 ZooKeeper 还支持一系列其他内置模式。表 6-6 描述了可用模式，以及 ACL 中使用的 ID 的格式。对于 `world` 模式，唯一的 ID 就是字符串 `anyone`。`digest` 模式使用 < 用户名 >:< 密码 > 的 sha1 值的 base64 编码。`ip` 模式允

许通过某个 IP 地址或 CIDR 标记的某段 IP 地址设置 ACL。最后，`sasl` 模式使用 < 主体 > 作为 ID，默认情况下，该主体即是用户的完整 UPN。可以通过设置 `kerberos.removeRealmFromPrincipal` 和 / 或 `kerberos.removeHostFromPrincipal` 控制主体名规范，这两项分别表示在比较 ID 之前移除域和移除第二部分。

表6-6：ZooKeeper模式

模式	ACL格式	描述
world	[illegible]	代表任意用户
digest	[illegible]:base64(sha1sum(< 用户名 >:< 密码 >))	代表使用密码认证的用户
ip	[illegible]	使用客户端 IP 作为身份标识
sasl	[illegible]	代表使用 SASL 认证的用户（例如 Kerberos 用户）

6.5 Oozie授权

Apache Oozie 的授权模型很简单，只有两个级别的账户：用户（**user**）和管理员用户（**admin user**）。用户具有如下权限：

- 所有作业的读权限
- 自身作业的写权限
- 对作业的基于单作业访问控制列表（用户和组列表）的写权限
- 管理操作的读权限

管理员用户具有如下权限：

- 所有作业的写权限
- 管理操作的写权限

可以设置 `oozie-site.xml` 文件中的如下参数，以启用 Oozie 授权：

```
<property>
  <name>oozie.service.AuthorizationService.security.enabled</name>
  <value>true</value>
</property>
<property>
```

```
  <name>oozie.service.AuthorizationService.admin.groups</name>
  <value>oozie-admins</value>
</property>
```

如果没有设置 `oozie.service.AuthorizationService.admin.groups` 参数，可以在 `adminusers.txt` 文件中指定管理员用户列表，每行一个：

```
oozie
alice
```

除了所有者和管理员用户拥有对作业的写权限，也可以使用针对作业的访问控制列表授予用户写权限。Oozie ACL 使用的语法与 Hadoop ACL 的一样（见表 6-1），提交作业时，通过工作流、协调器（coordinator）或 `job.properties` 包文件的 `oozie.job.acl` 属性进行设置。

6.6 HBase和Accumulo的授权

Apache HBase 和 Apache Accumulo 是基于 Google 的 BigTable 设计的，建立在 HDFS 和 ZooKeeper 之上的排序的、分布式的键 / 值存储。这两个系统有着类似的数据模型，也都是为在 HDFS 上实现随机化访问和更新负载而设计的。数据存储在行里，每行包含一个或多个列。和关系型数据库不同的是，每行包含的列可以是不同的。这方便在数据记录的模式不尽相同时，实现复杂的数据模型。每行都通过名为行 **id** 或行键的主键进行检索，每一行中，每个值又进一步通过列键和时间戳进行检索。行键、列键、时间戳以及它们所指向的数据合在一起称为单元（**cell**）。HBase 和 Accumulo 内部将数据作为排序的键值对进行存储，其中键由行 ID、列键和时间戳组成。列键又分为两部分：列家族（**column family**）和列限定符（**column qualifier**）。HBase 中，同一个列家族的所有列被存储在单独的磁盘文件中。而 Accumulo 中，不同的列家族可以被合并到本地组（**locality group**）。

一系列排序的行称为“表”。HBase 中，列家族的集合是要预先为每个表定义好的。而 Accumulo 允许用户动态创建新的列家族。两个系统中，列限定符都不需要预先定义，任意限定符都可以被插入任意行。表的逻辑组（类似于数据库或者关系型数据库系统中的模式）称

为“命名空间”。HBase 和 Accumulo 都支持系统、命名空间和表级别的权限，可用的权限及其含义各不相同，下面先看一下 Accumulo 的权限模型。

6.6.1　系统、命名空间和表级授权

Accumulo 最高支持系统级权限。通常来说，系统权限是为 Accumulo root 用户或 Accumulo 管理员预留的。高级别设置的权限会被低级别的对象所继承。例如，如果具有系统权限 `CREATE_TABLE`，则可以在任何命名空间创建表，即使没有显式地具有在该命名空间创建表的权限。表 6-7 给出了一系列系统级权限和描述，以及对应的命名空间级权限。

> 你将在本节看到多处提及 Accumulo root 用户，这与 root 系统账户是两码事。 Accumulo root 用户是在 Accumulo 初始化时自动创建的，该用户被授予所有系统级权限。root 用户不可能取消这些权限，以防无人能管理 Accumulo。

表6-7：Accumulo中的系统级权限

系统权限	描述	对应的命名空间权限
System.GRANT	向其他用户授予权限的权限；为 Accumulo root 用户预留	Namespace.ALTER_NAMESPACE
System.CREATE_TABLE	创建表的权限	Namespace.CREATE_TABLE
System.DROP_TABLE	删除表的权限	Namespace.DROP_TABLE
System.ALTER_TABLE	修改表的权限	Namespace.ALTER_TABLE
System.DROP_NAMESPACE	删除命名空间的权限	Namespace.DROP_NAMESPACE
System.ALTER_NAMESPACE	修改命名空间的权限	Namespace.ALTER_NAMESPACE
System.CREATE_USER	创建新用户的权限	N/A
System.DROP_USER	删除用户的权限	N/A
System.ALTER_USER	修改用户密码、权限和授权的权限	N/A
System.SYSTEM	对表或用户执行管理操作的权限	N/A
System.CREATE_NAMESPACE	创建新命名空间的权限	N/A

命名空间是表的逻辑集合，有助于管理表以及将管理功能委托给较小的用户组。假设市场部门需要一系列 Accumulo 表以支撑其一些应用，为了减轻 Accumulo 管理员的负担，可以创建一个 marketing 命名空间，然后将 `GRANT`、`CREATE_TABLE`、`DROP_TABLE` 和 `ALTER_TABLE` 权限授予 marketing 中的一个管理员。这会允许该部门无需系统级权限或等待 Accumulo 管理员操作，即可创建和管理它自己的表。很多命名空间级权限会被命名空间中的表继承，表 6-8 给出了一系列命名空间级权限和描述，以及对应的表级权限。

表6-8：Accumulo中的命名空间级权限

权限	描述
Namespace.READ	读（scan）命名空间中的表的权限
Namespace.WRITE	写（put，delete）命名空间中的表的权限
Namespace.GRANT	向命名空间中的表授予权限的权限
Namespace.BULK_IMPORT	向命名空间中的表批量导入数据的权限
Namespace.ALTER_TABLE	对命名空间中的表设置属性的权限
Namespace.DROP_TABLE	删除命名空间中的表的权限
Namespace.CREATE_TABLE	在命名空间中创建表的权限
Namespace.ALTER_NAMESPACE	设置命名空间属性的权限
Namespace.DROP_NAMESPACE	删除命名空间的权限

表级权限用于控制对单个表的粗粒度访问。表 6-9 给出了一系列表级权限及其描述。

表6-9：Accumulo中的表级权限

权限	描述
Table.READ	读取（scan）表的权限
Table.WRITE	写（put，delete）表的权限
Table.BULK_IMPORT	向表中批量导入数据的权限
Table.ALTER_TABLE	设置表的属性的权限
Table.GRANT	授予表权限的权限
Table.DROP_TABLE	删除表的权限

可以用 Accumulo 命令行管理系统、命名空间和表级权限。特别地，使用 `grant` 命令可以授予权限，使用 `revoke` 命令可以撤销权限。示例 6-18 使用 Accumulo 命令行管理权限。

示例 6-18　使用 Accumulo 命令行管理权限

```
root@cloudcat> userpermissions -u alice
System permissions:

Namespace permissions (accumulo): Namespace.READ

Table permissions (accumulo.metadata): Table.READ
Table permissions (accumulo.root): Table.READ
root@cloudcat> user alice
Enter password for user alice: *****
alice@cloudcat> table super_secret_squirrel
alice@cloudcat super_secret_squirrel> scan
2014-03-31 16:11:06,828 [shell.Shell] ERROR: java.lang.RuntimeException:
  org.apache.accumulo.core.client.AccumuloSecurityException: Error PERMISS
```

```
  for user alice on table super_secret_squirrel(ID:a) - User does not have
  to perform this action
alice@cloudcat super_secret_squirrel> user root
Enter password for user root: ******
root@cloudcat super_secret_squirrel> grant Namespace.READ -ns "" -u alice
root@cloudcat super_secret_squirrel> user alice
Enter password for user alice: *****
alice@cloudcat super_secret_squirrel> scan
r f:c []    value
alice@cloudcat super_secret_squirrel>
```

HBase 在系统、命名空间和表级使用相同的 ACL 权限集（表 6-10）。高级别设置的权限会被低级别的对象继承。例如，如果授予一个用户系统级 READ 权限，则该用户可以读取集群的所有表。除了向单个用户授予权限之外，HBase 还支持向用户组授予权限。使用 grant 命令时，在组名前加上前缀 `@`，以分配组权限。HBase 使用与 Hadoop 中相同的用户到组的映射类，组映射默认情况下会加载 HBase Master 上的 Linux 组，也支持 LDAP 组或自定义映射。

表6-10：HBase中的权限

权限	描述
READ(R)	读（get、scan）权限
WRITE(W)	写（put、delete）权限
EXEC(X)	执行协处理器终端的权限
CREATE(C)	删除表、修改表属性以及添加、修改、删除列家族的权限
ADMIN(A)	启用和禁用表、触发区域重分配和迁移的权限以及通过 CREATE 授予的权限

示例 6-19 描述了使用系统级权限授予对所有表的读权限。首先 Alice 打开 HBase 命令行，获取表的列表，然后尝试读取 `super_secret_squirrel` 表。

示例 **6-19**　Alice 被拒绝访问一个 HBase 表

```
[alice@cdh5-hbase ~]$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.0, rUnknown, Fri Feb  7 12:26:17 PST 2014

hbase(main):001:0> list
TABLE
super_secret_squirrel
1 row(s) in 2.2110 seconds
hbase(main):002:0> scan 'super_secret_squirrel'
ROW                                                  COLUMN+CELL
```

```
ERROR: org.apache.hadoop.hbase.security.AccessDeniedException: Insufficien
  permissions for user 'alice' for scanner open on table super_secret_squi

hbase(main):003:0> user_permission
User                                          Table,Family,Qualifier:Permiss
0 row(s) in 0.7350 seconds
```

要注意的是，Alice 执行 `user_permission` 命令时，她看不到任何人。Alice 要求 HBase 管理员授予其访问 HBase 中所有表的权限，管理员以 hbase 用户的身份登录到 HBase 命令行，然后使用 `grant` 命令授予 Alice 系统级的 READ 权限。

示例 **6-20**　HBase 管理员授予 Alice 系统级 READ 权限

```
[hbase@cdh5-hbase ~]$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.96.1.1-cdh5.0.0-beta-2, rUnknown, Fri Feb  7 12:26:17 PST 2014

hbase(main):001:0> grant 'alice', 'R'
0 row(s) in 2.6990 seconds

hbase(main):002:0> user_permission 'super_secret_squirrel'
User                                         Table,Family,Qualifier:Permissi
 hbase                                       super_secret_squirrel,,:[Permis
 actions=READ,WRITE,EXEC,CREATE,ADMIN]
1 row(s) in 0.2140 seconds
```

注意，由于 Alice 被授予的是系统级的权限，所以她仍然没有针对 `super_secret_squirrel` 表的权限。系统级的权限作为应用到 `hbase:acl` 表的内容，被展示在命令行里，如示例 6-21 所示。现在，当 Alice 执行 scan 动作，她就会得到表的行信息。

示例 **6-21**　Alice 现在可以扫描任意 HBase 表

```
hbase(main):004:0> user_permission
User                                          Table,Family,Qualifier:Permiss
 alice                                        hbase:acl,,: [Permission: acti
1 row(s) in 0.1540 seconds

hbase(main):005:0> scan 'super_secret_squirrel'
ROW                                                COLUMN+CELL
 r                                                 column=f:q, timestamp=1396
value=value
```

```
1 row(s) in 0.1310 seconds
```

6.6.2　列级别和单元级别授权

HBase 和 Accumulo 也支持数据级别的细粒度授权。HBase 中，可以在列级别指定权限。只有 `READ` 和 `WRITE` 权限可以应用到列级别 ACL。由于 HBase 支持分配权限到用户组，因此这是一种基于角色的访问控制（角色被一一映射到组）。HBase 模型与 Sentry 类似，也使用 RBAC，权限被存储在元数据层，而且在用户尝试访问表或列时得到应用。Accumulo 没有使用 RBAC 方式，而使用一种基于属性的安全。基于属性的安全使用标签对数据进行标记，并通过比较标签与用户的授权决定该用户是否具有读取数据值的权限。

Accumulo 中，安全标签存储在单元级，每个键值对都具有自己的标签。Accumulo 向 BigTable 的数据模型增加一个位于列限定符和时间戳中间的 `visibility` 元素，从而将安全标签作为键的一部分进行存储。和 Accumulo 中键的所有元素一样，安全标签不需要预先定义，可以在数据被插入时动态创建。为了支持更复杂的权限组合，安全标签包含一组使用布尔操作符 | 和 & 组合起来的自定义令牌，也可以使用括号制定布尔操作符的优先级。

除了和数据一起存储的标签之外，每个 Accumulo 用户还有一组安全标签。这些标签扫描数据时会与布尔表达式进行比较，然后过滤用户没有授权查看的单元。由于标签存储在单元级并且组成了键的一部分，所以实现多级安全很简单：同一行和列键可以指向不同授权级别的数据。这对于收集多个部分组成的相关数据（数据记录的一部分对所有用户公开，但更多的敏感部分是受限的）的公司来说，是个很强大的功能。

6.7　小结

本章介绍了允许或拒绝访问集群中数据或服务的授权机制。设置权限和 ACL 以控制对数据和资源的访问，是 Hadoop 管理的基础。我们看到不同组件的授权控制有所不同，尤其看到了授权访问数据的组件（HDFS、HBase、Accumulo）和授权访问计算和资源的组件（MapReduce、YARN）之间的区别。

至此，我们在单组件基础上实施的独立控制方面对授权进行了处理。虽然这对于限定对单个组件的访问很有效，但它增加了复杂度和管理

员的负担，管理员需要学习这些不同的控制方法。第 7 章将介绍随着 Apache Sentry（孵化中）的引入，授权控制是如何趋于统一的。

第 7 章　Apache Sentry（孵化中）

Hadoop 生态系统项目的整个生命周期中，安全授权以多种形式被添加进系统。对于系统管理员来说，实现和维持一个包含多个组件的通用授权系统变得越来越有挑战性。使问题更复杂的是，这些多样化组件的授权控制机制拥有不同等级的粒度和执行策略，这经常使管理员对于用户实际能在 Hadoop 环境中做什么（或不做什么）感到困惑。同其他问题一样，这些问题是提出 Apache Sentry（孵化中）的驱动力。

为了使管理员可以对用户能访问什么进行更灵活的控制，Sentry 项目（http://wiki.apache.org/incubator/SentryProposal）提出了对细粒度、基于角色的访问控制（**role-based access controls**，**RBAC**）机制的需求。通常，HDFS 的授权控制被限制为简单的 POXIS 权限和扩展ACL。那么，类似于 Hive、Cloudera Impala、Solr、HBase 和其他这种在 HDFS 顶层工作的框架，是怎样实现的呢？ Sentry 的目标是为 Hadoop 生态系统组件用统一的方法实现授权机制，这样安全管理员能够轻松地对用户和组所能访问的内容进行控制，而无需知道每一个 Hadoop 组件具体的输入和输出。

7.1　Sentry概念

每一个使用 Sentry 作为授权机制的组件必须与 Sentry 绑定，该绑定是组件将授权决策委托给 Sentry 的插件。为了执行授权决策，它会应用相关模型。例如，SQL 模型能够应用于 Hive 和 Impala，Search 模型能够用于 Solr，BigTable 模型能用于 HBase 和 Accumulo。稍后将会详细讨论 Sentry 的特权模型。

通过引用恰当的模型，Sentry 利用策略引擎访问策略提供者，以决定请求的行为是否获得了授权。策略提供者是策略的存储机制，例如数据库或文本文件。图 7-1 为 Sentry 组件的概念性视图。

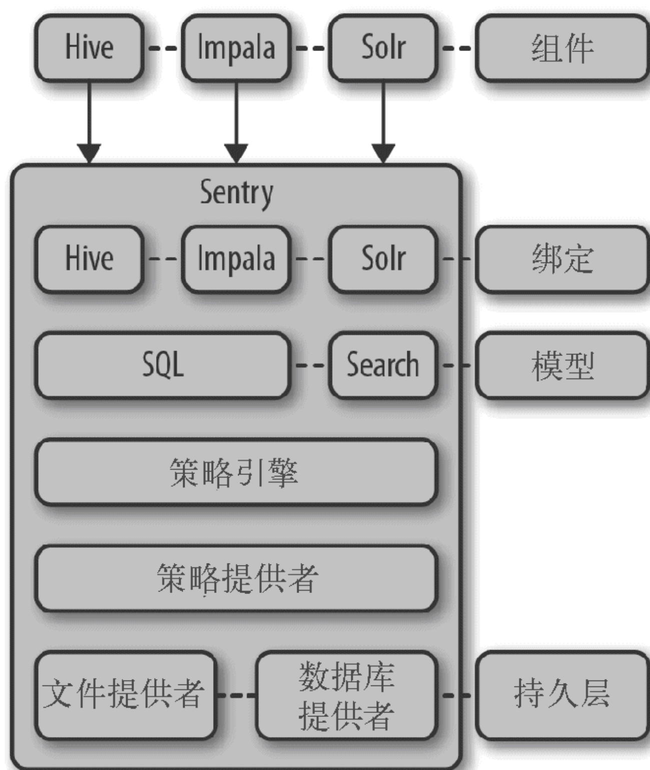


图 **7-1**：**Sentry** 组件

这个流程解释了组件如何从高层使用 Sentry，但策略引擎实际上的授权决策过程是怎样的呢？无论给定组件使用的模型如何，都有一些通用的关键概念。用户正是你所期望的样子，他们是执行某一特定行为的身份个体，这些行为可能是执行 SQL 查询、搜索某一集合、读取文件，或是检索一个键值对。用户也属于组。Sentry 的上下文中，组是拥有同样需求和特权的用户集合。Sentry 中的特权是一个数据访问单元，表示为一个包含对象和对该对象操作的元组。例如，对象可以是数据库、表或集合，操作可以是创建、读和写。

> Sentry 特权通常以肯定的方式进行定义，因为默认情况下，Sentry 会拒绝对每一个对象的访问。这种特权不能与稍后讲到的 `REVOKE` 语法混淆，REVOKE 功能仅仅是移除肯定特权。

最后，角色是特权的集合，是 Sentry 中准予授权的基本单元。一个

角色通常匹配一个业务功能，例如市场分析员或数据库管理员。用户、组、特权和角色之间的关系在 Sentry 中很重要，并且与下列逻辑紧密相关：

- 含有多个用户的组
- 被分配至一个组的角色
- 被授予特权的角色如图 7-2 所示。

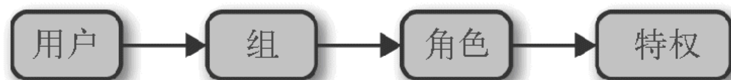


图 **7-2**：**Sentry** 实体间的关系

这种关系在 Sentry 中是严格强制执行的。例如，我们不可能将角色分配给一个用户，或是将特权授予一个组。该关系为严格执行时，会产生几种多对多关系。一个用户能够属于多个组，并且一个组可以包含多个用户。例如，Alice 可以同时隶属于市场组和开发组，并且开发组能够包含 Alice 和 Bob。

同样，一个角色能被分配给多个组，并且一个组能包含多个角色。例如，SQL 分析员角色能够同时被分配给市场组和开发组，并且开发组能够包含 SQL 分析员和数据库管理员两种角色。

最后，一个角色能够被授予多个特权，且某个特权能够成为多个角色的一部分。例如， SQL 分析员角色可以拥有点击流表中的 `SELECT` 特权和市场数据库中的 `CREATE` 特权，且同样的市场数据库中的 `CREATE` 特权也可以同时被赋给数据库管理员的角色。

了解 Sentry 高层概念后，下面继续深入了解实现环节。先从最新和最重要的开始：Sentry 服务。

7.2 Sentry服务

Sentry 项目首次进入 Apache 培育计划时，发布的第一个公开版本利用了基于插件的方法。能够利用 Sentry 的服务被配置 Sentry 插件（绑定模块），并且该插件运行于被配置的服务内部，直接读取策略文件。与许多 Hadoop 生态系统组件不同，Sentry 没有守护进程。此外，Sentry 策略在纯文本文件中进行配置，该文件枚举每一个策略。

每当添加、修改或删除一个策略时，都需要对文件进行修改。可以想见，这种方法维护起来相当简朴和笨拙，并且很容易出错。使问题更严重的是，策略文件中的错误会使整个文件都失效！

幸运的是，Sentry 已经极大地超越了早期的这种方法，并且在 Hadoop 生态系统中已经成长为“一等公民”。从版本 1.4 开始，Sentry 配备了一个能够被 Hive 和 Impala 利用的服务。该服务使用一个数据库后端代替了文本文件，以存储策略。另外，使用 Sentry 的服务被配置了指向 Sentry 的绑定，而不是采用在本地处理所有授权决策的绑定。由于 Sentry 架构的进步，除了老旧的系统版本之外，不推荐 Hive 和 Impala 采用基于策略文件的配置。即使这样，本章还是会对两种配置的相关内容都进行讲解。图 7-3 描述了 Sentry 服务是怎样融入 SQL 访问的。

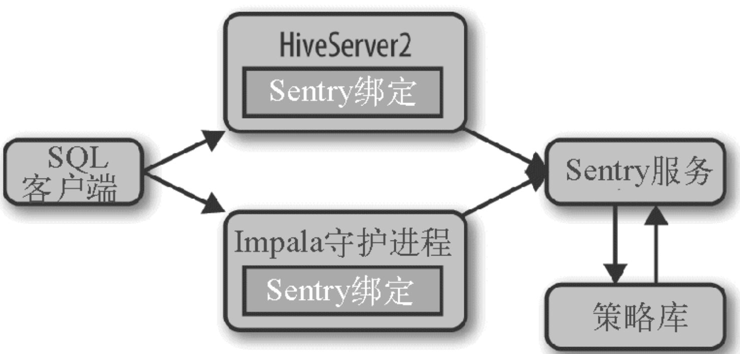


图 **7-3**：**Sentry** 服务架构

写作本书时，Solr 仍然在使用策略文件。可以预见，Solr 将会和其他新的启用 Sentry 的服务一样，抛弃基于策略文件的配置。

Sentry服务配置

配置 Sentry 并且在集群中运行的第一步是，要配置 Sentry 服务。Sentry 的主配置文件是 `sentry-site.xml`。示例 7-1 展示了在开启 Kerberos 的集群中对 Sentry 服务的一种典型配置，表 7-1 列举了配置参数。本章后续部分将讲解 Hadoop 生态系统组件是如何将 Sentry 服务用于授权的。

示例 **7-1**　Sentry 服务 sentry-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<configuration>
  <property>
    <name>sentry.service.server.rpc-address</name>
    <value>server1.example.com</value>
  </property>
  <property>
    <name>sentry.service.server.rpc-port</name>
    <value>8038</value>
  </property>
  <property>
    <name>sentry.service.admin.group</name>
    <value>hive,impala,hue</value>
  </property>
  <property>
    <name>sentry.service.allow.connect</name>
    <value>hive,impala,hue</value>
  </property>
  <property>
    <name>sentry.store.group.mapping</name>
    <value>org.apache.sentry.provider.common.HadoopGroupMappingService</va
  </property>
  <property>
    <name>sentry.service.server.principal</name>
    <value>sentry/_HOST@EXAMPLE.COM</value>
  </property>
  <property>
    <name>sentry.service.security.mode</name>
    <value>kerberos</value>
  </property>
  <property>
    <name>sentry.service.server.keytab</name>
    <value>sentry.keytab</value>
  </property>
  <property>
    <name>sentry.store.jdbc.url</name>
    <value>jdbc:mysql://server2.example.com:3306/</value>
  </property>
  <property>
    <name>sentry.store.jdbc.driver</name>
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property>
    <name>sentry.store.jdbc.user</name>
    <value>sentry</value>
  </property>
  <property>
    <name>sentry.store.jdbc.password</name>
    <value>sentry_password</value>
  </property>
</configuration>
```

表 7-1 展示了 `sentry-site.xml` 相关的所有配置参数，包括用于配置 Sentry 服务的参数、用于配置基于策略文件实现的参数，以及组件特定配置参数。

表7-1：sentry-site.xml配置

配置	描述
`sentry.provider`	对于 Hadoop 组来说通常为 `org.apache.sentry.provider.file.HadoopGroupResourceAuthorizationProvider`；本地组的该参数仅能在策略文件部署中被定义为 `org.apache.sentry.provider.file.LocalGro-upResourceAuthorizationProvider`
`sentry.provider.resource`	策略文件位置，可以为 `file://` 和 `hdfs://URIs`
`sentry.service.server`	Sentry 服务的名称，可以为任意名称
`sentry.provider.backend`	Sentry 服务类型：`org.apache.sentry.provider.file.SimpleFileProviderBackend` 或者 `org.apache.sentry.provider.db.SimpleDBProviderBackend`
`sentry.metastore.service.users`	针对 Hive 的 metastore，能够绕过 Sentry 策略的用户列表；只适用于 Senry 服务部署。
`sentry.provider`	与 `hive.sentry.provider` 相同的选项；用于 Solr
`sentry.service.admin.group`	用逗号间隔的组列表，是 Sentry 服务器的管理员
`sentry.service.allow.connect`	用逗号间隔的允许连接的用户列表；通常只是服务用户，不是终端用户
`sentry.service.client.server.rpcaddress`	Sentry 服务终端的客户端配置
`sentry.service.client.server.rpcport`	Sentry 服务端口的客户端配置
`sentry.service.security.mode`	Sentry 服务器上在处于的运行模式；可选 Kerberos 或无。
`sentry.service.server.keytab`	包含 `sentry.service.server.principal` 证书的 keytab 文件名
`sentry.service.server.principal`	`sentry.service.server.keytab` 包含的服务主体名，用于 Senry 服务的身份标识
`sentry.service.server.rpc-address`	启动 Sentry 服务的主机名
`sentry.service.server.rpc-port`	监听的端口
`sentry.solr.provider.resource`	Solr 中策略文件的位置；既可以为 `file://`，也可以为 hdsf://URIs
`sentry.store.jdbc.driver`	用于连接数据库的 JDBC 驱动名
`sentry.store.jdbc.password`	连接时使用的 JDBC 口令
`sentry.store.jdbc.url`	Sentry 服务器连接后端数据库时使用的 JDBC URL
`sentry.store.jdbc.user`	连接时使用的 JDBC 用户名
`sentry.store.group.mapping`	提供用户到组的映射的类，通常为 `org.apache.sentry.provider.common.HadoopGroupMappingService`

7.3　Hive授权

Sentry 的典型实现是，向 Hive 加入基于角色的访问控制。没有强授权控制机制，Hive 用户可以在没有太多限制的情况下对 Hive 的 metastore 进行变更。另外，对 Hive 表的访问完全受底层 HDFS 文件系统访问许可的控制，该访问许可非常受限。通过 Sentry，安全管理员能够非常细粒度地控制哪些用户和组能使用 Hive，包括创建表和视图、向已存在的表插入新数据或查询数据等操作。

为了理解 Sentry 是如何向 Hive 提供授权机制的，首先需要理解 Hive 组件是如何合作的。Hive 架构中有 3 个主要组件：metastore 数据库、Hive Metastore Server 和 HiveServer2。metastore 数据库

是一个关系型数据库，包括数据库、表和视图信息、表数据在 HDFS 中的位置、列的数据类型、文件格式和压缩等 Hive 元数据。当一个客户端与 Hive 互动时，这些信息对于理解即将执行的操作是十分必要的。

Hive 的早期版本中，这些差不多是用户能拥有的全部；Hive 客户端 API 会直接与 metastore 数据库对话并执行操作。从安全角度来说，这是件坏事。这个模型意味着每一个 Hive 客户端都拥有访问 Hive 元数据库的全部权限！ Hive Metastore Server 成为 Hive 架构下解决这个问题的组件。该角色的目的是成为 Hive 客户端和 metastore 数据库之间的中间层。通过这个模型，客户端只需要知道怎样与 Hive Metastore Server 联系，只有 Hive Metastore Server 负责与下层 metastore 数据库交互。

Hive 架构中的最后一个组件是 HiveServer2。这个组件的目的是通过 JDBC 和 ODBC 这类接口，向外部应用提供查询服务。HiveServer2 从客户端接收请求，与 Hive Metastore Server 通信，以取回元数据信息，并执行恰当的 Hive 操作，例如分发 MapReduce 作业。顾名思义，HiveServer2 是这种服务的第二个版本——第一版缺乏并发性和安全特性。理解这部分的重要之处在于，HiveServer2 最初就是用于向外部应用提供服务的。Hive 命令行接口（CLI）仍然与 Hive Metastore Server 直接交互，并使用 Hive API 执行操作。用户可以在 HiveServer2、`beeline` 中使用 CLI 执行操作，但这不是必须的。这个事实向面向所有客户端的强制授权机制提出了挑战。你可能猜到了，为了实现这个目标（强制授权），就要加强 HiveServer2 的安全授权，并确保所有 SQL 客户端都必须通过 HiveServer2 执行任何 Hive 的 SQL 操作。

Hive 架构中的另一个组件是 HCatalog。这是一个库的集合，该集合中的库能够允许非 SQL 客户端访问 Hive Metastore 结构体。这对于 Pig 或 MapReduce 用户很有用，他们可以不使用传统的 Hive 客户端确定文件的 metadata 结构体。HCatalog 库的一个扩展是 WebHCatServer 组件，该组件是一个提供 REST 接口以执行 HCatalog 函数的守护进程。

HCatalog 库和 WebHCatServer 都不利用 HiveServer2。所有通信都是直接与 Hive Metastore Server 进行的。由于这一点，Hive Metastore Server 也必须受到 Sentry 的保护，以确保 HCatalog 用户不能对 Hive Metastore 数据库进行任意修改。

 虽然 Sentry 的 1.4 版能够对 Hive Metastore

Server 提供写保护，但它并不同时限制读操作。这意味着，某个用户在 HCatalog 中进行一些相当于 SHOW TABLES 的操作时，系统会返回所有表的列表，包括用户没有权限访问的那些表。这与通过 HiveServer2 执行的相同操作不同，HiveServer2 中，用户只能看见他们有权限访问的那些对象。然而这仅仅是元数据泄漏。实际数据的访问在 HDFS 访问的时候仍旧会被强制控制。如果集群没有任何使用 HCatalog 的用户，那么对所有 Hive 和 HiveServer2 的流量实现强制访问控制的方法是，在配置文件 core-site.xml 中将属性 hadoop.proxyuser.hive.groups 设置为 hive，impala，这样就能允许有且仅有 Hive（HiveServer2）和 Impala（Catalog Server）都能直接访问 Hive Metastore Server。

图 7-4 展示了 Hive 架构的布局和 Sentry 的强制访问机制的发生位置。无论什么访问方法，核心的强制机制是保护 Hive 元数据不受非授权修改。

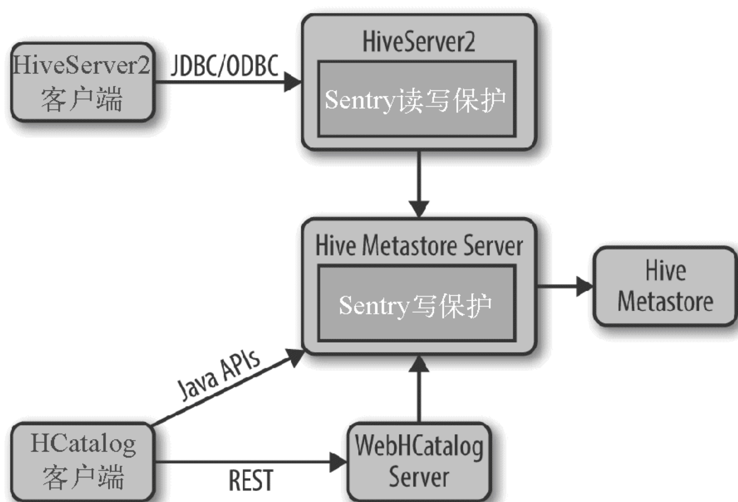


图 7-4：**Hive sentry** 架构

Hive Sentry的配置

本节将了解通过配置 Hive 使用 Sentry 授权的必需要素。示例 7-2 展

示了能够利用 Sentry 授权的，用于 Hive Metastore Server 和 HiveServer2 的配置文件 sentry-site.xml。

示例 **7-2** Hive sentry-site.xml 服务部署

```
<configuration>
  <property>
    <name>hive.sentry.server</name>
    <value>server1</value>
  </property>
  <property>
    <name>sentry.service.server.principal</name>
    <value>sentry/_HOST@EXAMPLE.COM</value>
  </property>
  <property>
    <name>sentry.service.security.mode</name>
    <value>kerberos</value>
  </property>
  <property>
    <name>sentry.hive.provider.backend</name>
    <value>org.apache.sentry.provider.db.SimpleDBProviderBackend</value>
  </property>
  <property>
    <name>sentry.service.client.server.rpc-address</name>
    <value>server1.example.com</value>
  </property>
  <property>
    <name>sentry.service.client.server.rpc-port</name>
    <value>8038</value>
  </property>
  <property>
    <name>hive.sentry.provider</name>
    <value>
  org.apache.sentry.provider.file.HadoopGroupResourceAuthorizationProvider
  </value>
  </property>
  <property>
    <name>sentry.metastore.service.users</name>
    <value>hive,impala,hue,hdfs</value>
  </property>
</configuration>
```

示例 7-3 展示了使用 HiveServer2（与 Hive Metastore Server）和基于策略文件的部署时，Sentry 的一种典型配置。Sentry 中，基于策略文件的配置比基于服务的配置精简许多，但它们之间仍然存在共同点。稍后将看到，HiveServer2 守护进程的 hive-site.xml 配置文件中，指明了 sentry-site.xml 文件在本地文件系统中的位置。

示例 **7-3** Hive sentry-site.xml 策略文件部署

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <property>
    <name>hive.sentry.provider</name>
    <value>
      org.apache.sentry.provider.file.HadoopGroupResourceAuthorizationProv
    </value>
  </property>
  <property>
    <name>sentry.hive.provider.backend</name>
    <value>org.apache.sentry.provider.file.SimpleFileProviderBackend</valu
  </property>
  <property>
    <name>hive.sentry.provider.resource</name>
    <value>/user/hive/sentry/sentry-provider.ini</value>
  </property>
  <property>
    <name>hive.sentry.server</name>
    <value>server1</value>
  </property>
</configuration>
```

首先看看两个例子中都用到的配置属性。`hive.sentry.server` 参数为这个特定的 Sentry 服务器指定了名称（标签），可以在策略中引用。这个名称与机器的主机名无关。配置 `sentry.hive.provider.backend` 可以告诉 Hive 使用哪个后端服务。参数 `org.apache.sentry.provider.db.SimpleDBProviderBackend`（S 服务）和 `org.apache.sentry.provider.file.SimplefileProviderBacker` 策略文件）分别配置了 Sentry 确定组信息的方法。这里展示的 `HadoopGroupResourceAuthorizationProvider` 将会利用 hadoop 的所有配置方法，如从本地操作系统读取组，或者从 LDAP 上直接拉取组信息等。然而这仅是示例 7-2 举出的一种形式，因为 Sentry 服务不能使用策略文件定义本地用户到组的映射。

接下来了解专门针对 Sentry 服务的配置示例。Hive 如何连接到 Sentry 服务的细节由以下参数提供：

- `sentry.service.client.server.rpc-address`
- `sentry.service.client.server.rpc-port`

sentry.service.server.principal 和 sentry.service.security.mode 都会对 Kerberos 配置的细节进行设置。最后，sentry.metastore.service.users 的配置会列出那些被允许能够绕过 Sentry 授权机制，并且与 Hive Metastore Server 直连的用户——通常是类似于 Hive 和 Impala 的服务 / 系统用户。

对 hive.sentry.provider.resource 的配置是特定于域策略文件部署示例的。该参数指定了策略文件的存储位置。指定的 Sentry 策略文件位置会假定与 hdfs-site.xml 文件中指定的位置一样。例如，若 hdfs-site.xml 指向 HDFS，就会默认 HDFS 中存在路径 user/hive/sentry/sentry-provider.ini。此外也可以通过提供 HDFS 路径 hdfs:// 或者本地文件系统路径 file:// 明确存储位置。

sentry-site.xml 的配置对 Hive 很重要，与之形成对比的是，该文件并未对其本身启用 Sentry 授权机制。因此，在 Hive 的配置文件 hive-site.xml 中进行额外配置是必要的。示例 7-4 展示了 Hive Metastore Server 需要的相关配置，与之类似，示例 7-5 展示了 HiveServer2 所需的相关配置。这两种配置虽然相似，但仍有所区别。示例 7-6 中，最后的 hive-site.xml 示例展示了 HiveServer2 在基于策略文件的部署中所需的配置。注意，基于策略文件的部署中，Hive Metastore Server 不需要额外配置（稍后详述）。

示例 **7-4**　Hive Metastore Server hive-site.xml 的 Sentry 服务配置

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <!-- Unrelated properties omitted -->
  <property>
    <name>hive.sentry.conf.url</name>
    <value>file:///etc/hive/conf/sentry-site.xml</value>
  </property>
  <property>
    <name>hive.metastore.pre.event.listeners</name>
    <value>org.apache.sentry.binding.metastore.MetastoreAuthzBinding</valu
  </property>
  <property>
    <name>hive.metastore.event.listeners</name>
    <value>
      org.apache.sentry.binding.metastore.SentryMetastorePostEventListener
    </value>
```

```
  </property>
</configuration>
```

示例 7-5　HiveServer2 hive-site.xml 的 Sentry 服务配置

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <!-- Unrelated properties omitted -->
  <property>
    <name>hive.server2.enable.doAs</name>
    <value>false</value>
  </property>
  <property>
    <name>hive.server2.session.hook</name>
    <value>org.apache.sentry.binding.hive.HiveAuthzBindingSessionHook</val
  </property>
  <property>
    <name>hive.sentry.conf.url</name>
    <value>file:///etc/hive/conf/sentry-site.xml</value>
  </property>
  <property>
    <name>hive.security.authorization.task.factory</name>
    <value>
      org.apache.sentry.binding.hive.SentryHiveAuthorizationTaskFactoryImp
    </value>
  </property>
</configuration>
```

示例 7-6　HiveServer2 hive-site.xml 的 Sentry 策略文件配置

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <!-- Unrelated properties omitted -->
  <property>
    <name>hive.server2.enable.doAs</name>
    <value>false</value>
  </property>
  <property>
    <name>hive.server2.session.hook</name>
    <value>org.apache.sentry.binding.hive.HiveAuthzBindingSessionHook</val
  </property>
  <property>
    <name>hive.sentry.conf.url</name>
    <value>file:///etc/hive/conf/sentry-site.xml</value>
```

```
  </property>
</configuration>
```

所有三个 `hive-site.xml` 示例中，配置参数 `hive.sentry.conf.url` 告诉 Hive 应到何处定位 Sentry 配置文件。HiveServer2 的两个示例中，参数 `hive.server2.session.hook` 用于指定一个绑定，该绑定会实质上将授权决策传递给 Sentry。

同样，HiveServer2 的两个示例中，要注意用户模拟功能是关闭的。关闭 Hive 的用户模拟是 Sentry 配置中的关键一环。为了真正实现从查询到数据访问这一过程的强制授权，Sentry 和 Hive 需要对查询接口和文件访问都进行控制。为此，Hive warehouse 的 HDFS 权限需要被锁定，如示例 7-7 所示。

示例 **7-7**　锁定 Hive 数据仓库

```
[alice@server1 ~]$ kinit hdfs
Enter password for hdfs@EXAMPLE.COM:
[alice@server1 ~]$ hdfs dfs -chown -R hive:hive /user/hive/warehouse
[alice@server1 ~]$ hdfs dfs -chmod -R 0771 /user/hive/warehouse
[alice@server1 ~]$
```

锁定 Hive warehouse 并且禁用用户模拟后，Sentry 便控制了查询接口的授权。由于只有 Hive 系统用户能够访问文件，因此 HDFS 权限也被锁定。从安全角度来说这样更好，而且这还使得 Sentry 拥有在低至视图级进行控制授权的能力。视图能被用于列级安全（只选择某些特定列）和行级安全，如提供一个 `WHERE` 子句的过滤器。如果启用用户模拟，那么查询行为会被当作终端用户。视图级权限没有被切合实际地强制限制，因为用户拥有 HDFS 的文件级（如表级）访问权限，并且能够通过直接访问文件绕过 Sentry 策略，例如 MapReduce。

7.4　Impala授权

Sentry 的初始版本包括对 Hive 和 Impala 的支持。虽然这两个组件拥有一些相似性，但它们在架构上有着本质区别，所以彻底理解 Sentry 如何融入系统之前，需要先对此进行强调。首先，Impala 是一整个处理框架。与 Hive 不同的地方在于，Hive 没有任何处理能力进行用户要求的实际工作。这种工作默认由 MapReduce 处理的（独

立的版本 1，或者 YARN 上的版本 2）。

Impala 架构由三部分组成：Daemon、StateStore 和 Catalog 服务。Impala Daemon 即 impalad，是实际上的工作进程，它运行于集群中每一个运行有 HDFS DataNode 守护进程的节点。Impala StateStore 即 statestored，负责记录集群中所有 impalad 实例的健康状态。若一个实例出现问题，statestored 会将这个信息广播给其余所有 impalad 实例。虽然这看起来像是 Impala 架构中的核心组件，但实际上它不是必需的。如果 statestored 进程挂掉或根本不存在，那么所有被 impalad 实例完成的工作会继续运作。唯一潜在的影响是，如果一个 impalad 实例变得不健康，则余下的实例会发现得很慢，这会导致总的查询执行时间延迟。Impala Catalog 服务又叫 catalogd，负责追踪记录元数据的变化。如果一个 Impala 查询在一个 impalad 上被执行，并以某种方式改变了元数据，catalogd 会将更新的元数据广播至其他 impalad 实例。catalogd 负责与 Hive Metastore Server 通信，以取得所有存在的元数据信息。

之前已经回顾了 Impala 架构的基础知识，下面学习 Sentry 真正发挥作用的地方。正如早先对 Hive 的讨论中阐述的那样，Sentry 是 Hive 组件 HiveServer2 和 Hive Metastore Server 的插件，大多数时候是集群中的独立实例。通过 Impala，Sentry 不再是用于扩充单一主组件的集中化插件，就像给 catalogd 或者 statestored 进程做的那样。实际上，Sentry 在每一个 impalad 实例中都被启用。用户通过 Sentry 连接某个特定 impalad 并进行一次查询时，impalad 利用 Sentry 策略（要么来自 Sentry 服务，要么来自策略文件）决定该用户是否被授权执行这次请求行为。

Impala的Sentry配置

正如前面学习 Hive 一样，本节将了解如何配置 Impala，使之利用 Sentry 进行授权控制。示例 7-8 展示了 Impala 守护进程为利用 Sentry 服务而使用的配置文件 sentry-site.xml。基于策略文件的部署中，sentry-site.xml 文件不是必需的。

示例 **7-8**　Impala sentry-site.xml 的服务部署

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <property>
    <name>sentry.service.server.principal</name>
    <value>sentry/_HOST@EXAMPLE.COM</value>
```

```
  </property>
  <property>
    <name>sentry.service.security.mode</name>
    <value>kerberos</value>
  </property>
  <property>
    <name>sentry.service.client.server.rpc-address</name>
    <value>server1.example.com</value>
  </property>
  <property>
    <name>sentry.service.client.server.rpc-port</name>
    <value>8038</value>
  </property>
</configuration>
```

你可能已经意识到，Impala 使用 Sentry 服务时，对 `sentry-site.xml` 的配置正是 Hive 同类配置的子集。上一节已经讨论了这些属性，现在可以直接看看要想启用 Sentry 授权机制，应如何配置 Impala 守护进程。

Sentry 服务的部署中，Impala 守护进程仅需要配置三个标识。第一个标识是 `server_name`，它是 Sentry 服务器的标签。该标识与 `hive.sentry.server` 的配置参数相匹配。第二个标识是 `sentry_config`，将 Impala 守护进程指向配置文件 `sentry-site.xml` 的存储位置。第三个标识是 `authorized_proxy_user_config`，用于指定某些用户作为其他用户的模拟，例如 hue 用户。示例 7-9 展示了这种配置。

示例 **7-9**　用于 Sentry 服务配置的 Impala 标识

```
...Other unrelated flags omitted for brevity
-server_name=server1
-sentry_config=/etc/impala/conf/sentry-site.xml
-authorized_proxy_user_config=hue=*
```

基于 Sentry 策略文件的部署中，Impala 守护进程不需要 `Sentry_Config` 标识，而配置为 `authorization_policy_file` 和 `authorization_policy_provider_class` 标识。这些标识分别指明了 Sentry 策略文件的位置和授权提供者类的位置。后者的配置属性已被设为 `hive.sentry provider`，可达到同样的目的，如示例 7-10 所示。

示例 7-10

```
...Other unrelated flags omitted for brevity
-server_name=server1
-authorized_proxy_user_config=hue=*
-authorization_policy_file=/user/hive/sentry/sentry-provider.ini
-authorization_policy_provider_class=\
  org.apache.sentry.provider.file.HadoopGroupResourceAuthorizationProvider
```

7.5　Solr授权

Solr 的授权始于集合。集合是访问的主要入口，集合之于 Sole 类似于数据库之于 SQL。Sentry 授权最开始是在集合层面定义特权的。Sentry 由此演变出文件级授权。文件级授权通过给每个文件标上特定的字段名实现，该字段名包括与在 Sentry 策略文件（稍后进行阐述）中定义的相关 Sentry 角色名对应的值。给文件打标签的步骤将会在数据导入时进行，因此，通过良好的角色名避免重新处理文件修改标签是很重要的。

Solr的Sentry配置

本节阐述如何给 Solr 配置 Sentry 授权。示例 7-11 展示了如何在配置文件 `sentry-site.xml` 中对 Solr 服务器进行配置。

示例 7-11　Solr 的 sentry-site.xml 配置文件部署

```
<?xml version="1.0" encoding="UTF-8"?>

<configuration>
  <property>
    <name>sentry.provider</name>
    <value>
      org.apache.sentry.provider.file.HadoopGroupResourceAuthorizationProv
    </value>
  </property>
  <property>
    <name>sentry.solr.provider.resource</name>
    <value>/user/solr/sentry/sentry-provider.ini</value>
  </property>
</configuration>
```

示例 7-11 展示的两个配置参数此时看起来非常相似，但这些参数名在 Solr 中仍然略有区别。配置参数 `sentry.provider` 与 Hive 中

的 hive.sentry.provider、Impala 中的 authorization_policy_provider_class 标识的工作原理相似。配置参数 sentry.solr.provider.resource 指定了 Sentry 策略文件的位置。重申一下，策略文件既可以放在本地文件系统，也可以放在 HDFS 中。该文件需要对正在运行 Solr 服务的用户（如 solr 用户）可读。

为了给 Solr 服务配置 Sentry 授权，需要一些环境变量。这些参数既可以配置为环境变量，也可以配置为 /etc/default/solr 配置文件中的参数行。第一个参数是 SOLR_SENTRY_ENABLED。显然，该参数被置为 ture 时，启用 Sentry 授权机制。下一个参数是 SOLR_AUTHORIZATION_SUPERUSER。该参数用于定义拥有超级用户权限的用户，即 solr 用户。最后一个参数为 SOLR_AUTHORIZATION_SENTRY_SITE，该参数指定了配置文件 sentry-site.xml 的位置（稍后介绍）。示例 7-12 展示了这些配置。

示例 **7-12**　Sentry 策略文件使用中的 Solr 环境变量

```
...Other unrelated environment variables omitted for brevity
SOLR_SENTRY_ENABLED=true
SOLR_AUTHORIZATION_SUPERUSER=solr
SOLR_AUTHORIZATION_SENTRY_SITE=/etc/solr/conf/sentry-site.xml
```

本节早先提到过，能够进行文件级授权。为了实现这个功能，有必要对集合进行一些配置。集合的配置默认通过配置文件 solrconfig.xml 进行。该文件如示例 7-13 所示。

示例 **7-13**　文件级安全 solrconfig.xml

```
<searchComponent name="queryDocAuthorization"
class="org.apache.solr.handler.component.QueryDocAuthorizationComponent">
    <bool name="enabled">true</bool>
    <str name="sentryAuthField">sentry_auth</str>
    <str name="allRolesToken">*</str>
</searchComponent>
```

示例 7-13 展示了 org.apache.solr.handler.component.QueryDocAuthorizationC
类被用于文件级授权决策，将 enabled 参数设为 true 即可开启。配置参数 sentryAuthfield 定义了文件包含能确定访问权限

的授权令牌的字段名称。senty_auth 的默认值如示例所示，但也可被设置为任意值。文件将会使用这个标签插入访问本文件所需的角色名。最后一个配置参数 allRolesToken 定义了一个允许所有角色都可访问某个文件的令牌。该参数默认值为 *，可以保持该值不变，从而与其他 Sentry 特权的通配符保持一致。

7.6 Sentry特权模型

本节将了解 Sentry 提供授权的多种服务的特权模型。特权模型能够标识特权、特权对应的对象类型、特权有效范围及其他有助于安全管理员理解可用授权控制的相关信息和粒度。

充分理解特权模型能够确保选择合适的策略，以满足授权控制需要的程度，并保护数据不会受到非授权的访问。

7.6.1 SQL特权模型

Sentry 为 SQL 访问提供了三种权限：SELECT、INSERT 和 ALL。这些权限并不可用于每一个对象。表 7-2 提供了 SQL 上下文中权限和对象的对应情况。SQL 权限模型是分层的，这意味着容器客体会将特权传递给子对象。这对于透彻理解用户是否拥有访问权限是很重要的。

表7-2：SQL特权类型[a]

特权	对象
INSERT	TABLE, URI
SELECT	TABLE, VIEW, URI
ALL	SERVER, DB, URI

a 所有特权模型表都由 Cloudera, Inc 授权，从 cloudera.com 复制而来。

表 7-3 列出了容器权限在给定对象处的受到粒度特权的支配。例如，表中第一行可理解为“一个 SERVER 对象的 ALL 权限使得 DATABASE 对象也拥有 ALL 权限”。

表7-3：SQL权限层级

容器对象	[illegible]	粒度权限	
SERVER / DATABASE [illegible]			

DATABASE			
DATABASE			
DATABASE			

SQL 特权模型的最后一部分是，理解权限如何映射到 SQL 操作。表 7-4 展示了某个给定的 SQL 操作、操作应用的对象范围和执行该操作需要什么权限。例如，表中第一行可理解为“CREATE DATABASE 操作可应用于 SERVER 对象并且需要对 SERVER 对象的 ALL 权限”。一些 SQL 操作需要不止一种权限，例如创建视图。创建一个新的视图需要有对视图所在 DATABASE 的 ALL 权限，也需要有对于被新视图引用的表／视图对象的 SELECT 权限。

表**7-4**：**SQL**权限

	SQL操作		
CREATE DATABASE			
DROP DATABASE			
CREATE TABLE			
DROP TABLE			
[illegible] SELECT on TABLE			
DROP VIEW			
CREATE INDEX			
DROP INDEX			
ALTER TABLE ADD COLUMNS			
ALTER TABLE REPALCE COLUMNS			
ALTER TABLE CHANGE column			
ALTER TABLE RENAME			
ALTER TABLE SET TBLPROPERTIES			
ALTER TABLE SET FILEFORMAT			
ALTER TABLER SET LOCATION			
ALTER TABLE ADD PARTITION			
ALTER TABLE ADD PARTITION location			
ALTER TABLE DROP PARTITION			
ALTER TABLE PARTITION SET FILEFORMAT			
SHOW TBLPROPERTIES			
SHOW CREATE TABLE			
SHOW PARTITIONS			
DESCRIBE TABLE			
DESCRIBE TABLE PARTITION			
LOAD DATA			
SELECT			
INSERT OVERWRITE TABLE			
CREATE TABLE AS SELECT			
USE database			
ALTER TABLE SET SERDEPROPERTIES			
ALTER TABLE PARTITION SET SERDEPROPERTIES			
CREATE ROLE			
GRANT ROLE TO GROUP			

GRANT PRIVILEGE ON SERVER	SERVER	ALL
GRANT PRIVILEGE ON DATABASE	DATABASE	WITH GRANT OPTION
GRANT PRIVILEGE ON TABLE	TABLE	WITH GRANT OPTION

虽然 Hive 和 Impala 支持大多数 SQL 操作，但仍有一些操作仅能被 Hive 或 Impala 一方支持，或均尚未实现。表 7-5 列出了仅能应用于 Hive 的 SQL 权限，表 7-6 列出了仅能应用于 Impala 的 SQL 权限。

表**7-5**：仅用于**Hive**的**SQL**权限

SQL操作	**范围**	**权限**
INSERT OVERWRITE DIRECTORY	TABLE	INSERT, URI
ANALYZE TABLE	TABLE	SELECT, INSERT
IMPORT TABLE	DATABASE	ALL, URI
EXPORT TABLE	TABLE	SELECT, URI
ALTER TABLE TOUCH	TABLE	ALTER
ALTER TABLE TOUCH PARTITION	TABLE	ALTER
ALTER TABLE CLUSTERED BY SORTED BY	TABLE	ALTER
ALTER TABLE ENABLE/DISABLE	TABLE	ALTER
ALTER TABLE PARTITION ENABLE/DISABLE	TABLE	ALTER
ALTER TABLE PARTITION RENAME TO PARTITION	TABLE	ALTER
ALTER DATABASE	DATABASE	ALL
DESCRIBE DATABASE	DATABASE	SELECT, INSERT
SHOW COLUMNS	TABLE	SELECT, INSERT
SHOW INDEXES	TABLE	SELECT, INSERT

表**7-6**：仅用于**Impala**的**SQL**权限

SQL操作	**范围**	**权限**
EXPLAIN	TABLE	SELECT
INVALIDATE METADATA	SERVER	ALL
INVALIDATE METADATA table	TABLE	SELECT/INSERT
REFRESH table	TABLE	SELECT/INSERT
CREATE FUNCTION	SERVER	ALL
DROP FUNCTION	SERVER	ALL
COMPUTE STATS	TABLE	ALL

7.6.2 Solr特权模型

对于 Solr，Sentry 提供三种权限：`QUERY`、`UPDATE` 和 `*`(ALL)。Solr 的特权模型分为应用于请求处理程序（handler）的特权和应用于集合的特权。表 7-8~表 7-10 中，`admin` 集合名在 Sentry 中是用于代表管理行为的特殊集合。所有 Solr 特权模型表中，

collection1 表示一个任意集合名。

表7-7：非管理事务请求处理程序的Solr权限表

		[illegible]		
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				

表7-8：admin集合操作的Solr权限表

		[illegible]		
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				

表7-9：核心admin操作的Solr权限表

		[illegible]		
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				
[illegible]				

replicationHandler	UPDATE	collection1
admin, ApplyUpdates	UPDATE	collection1

表7-10：信息和管理员处理程序的Solr权限表

处理程序	集合	需要的权限
LukeRequestHandler	admin, luke	QUERY
SystemInfoHandler	admin, system	QUERY
SolrInfoMBeanHandler	admin, mbeans	QUERY
PluginInfoHandler	admin, plugins	QUERY
ThreadDumpHandler	admin, threads	QUERY
PropertiesRequestHandler	admin, properties	QUERY
LoggingHandler	admin, logging	QUERY, UPDATE(or*)
ShowFileRequestHandler	admin, file	QUERY

7.7 Sentry策略管理

前面已经讲解了权限表和可用的访问方式，下面进一步了解实际的策略是怎样被添加、删除或修改的。管理不同策略的方法要依赖于 Sentry 的部署类型，要么是新的 Sentry 服务，要么是旧的策略文件。对于前者（这也是我们推荐的类型），管理策略文件的方法是通过 SQL 命令实现的。

7.7.1 SQL命令

习惯了在流行的关系型数据库系统中管理角色和许可的安全管理员会发现，管理 Sentry 策略的 SQL 语法十分眼熟。表 7-11 展示了管理员管理 Sentry 策略时的所有可用声明。

表7-11：Sentry策略的SQL语法

语法	描述
CREATE ROLE *role name*	创建拥有指定名称的角色
DROP ROLE *role name*	删除拥有指定名称的角色
GRANT ROLE *role name* TO GROUP *group name*	分配指定角色到指定的组
REVOKE ROLE *role name* FROM GROUP *group name*	从指定的组移出指定角色
GRANT *privilege* ON *object* TO ROLE *role name*	将对某指定对象的权限分配给指定角色
GRANT *privilege* ON *object* TO ROLE *role name* WITH GRANT OPTION	将对某指定对象的权限分配给指定角色并允许该角色获取更多该对象的权限
REVOKE *privilege* ON *object* FROM ROLE *role name*	取消指定角色的对指定对象的权限
SET ROLE *role name*	为当前会话设置指定角色
SET ROLE ALL	为当前会话启用（用户能访问的）所有角色
SET ROLE NONE	禁用当前会话的所有角色
SHOW ROLES	列出数据库中的所有角色

SHOW CURRENT ROLES	显示当前会话启用的所有角色
SHOW ROLE GRANT GROUP *group_name*	显示指定组的所有角色
SHOW GRANT ROLE *role_name*	显示指定角色的所有许可权限
SHOW GRANT ROLE *role_name* ON *object* *object_name*	显示指定角色关于指定对象的所有许可权限

表 7-11 提供了多种语法的列表，一个处于工作状态中的示例能保证看到这些行为（示例 7-14）。

示例 **7-14**　Sentry SQL 的使用示例

```
# Authenticated as the hive user, which is a member of a group listed in
# sentry.service.admin.group and accessing HiveServer2
# via the beeline CLI

# Create the role for hive administrators
0: jdbc:hive2://server1.example.com:100> CREATE ROLE hive_admin;
No rows affected (0.852 seconds)

# Grant the hive administrator role to the sqladmin group
0: jdbc:hive2://server1.example.com:100> GRANT ROLE hive_admin TO GROUP sc
No rows affected (0.305 seconds)

# Grant server-wide permissions to the hive_admin role
0: jdbc:hive2://server1.example.com:100> GRANT ALL ON SERVER server1
TO ROLE hive_admin;
No rows affected (0.339 seconds)

# Show all of the roles in the Sentry database
0: jdbc:hive2://server1.example.com:100> SHOW ROLES;

+-------------+
|     role    |
+-------------+
| hive_admin |
+-------------+
1 row selected (0.63 seconds)

# Show all the privileges that the hive_admin role has access to
# (some columns omitted for brevity)
0: jdbc:hive2://server1.example.com:100> SHOW GRANT ROLE hive_admin;
+-----------+-----------------+-----------------+------------+-----------
| database  | principal_name  | principal_type  | privilege  | grant_optic
+-----------+-----------------+-----------------+------------+-----------
| *         | hive_admin      | ROLE            | *          | false
+-----------+-----------------+-----------------+------------+-----------
+----------+
| grantor  |
+----------+
| hive     |
+----------+
```

```
1 row selected (0.5 seconds)

# Show all the roles that the sqladmin is a part of
0: jdbc:hive2://server1.example.com:100> SHOW ROLE GRANT GROUP sqladmin;
+-------------+--------------+-------------+---------+
|    role     | grant_option | grant_time  | grantor |
+-------------+--------------+-------------+---------+
| hive_admin  | false        |             | hive    |
+-------------+--------------+-------------+---------+
1 row selected (0.5 seconds)

# Remove all of the roles for the current user session
0: jdbc:hive2://server1.example.com:100> SET ROLE NONE;
No rows affected (0.2 seconds)

# Show list of current roles
0: jdbc:hive2://server1.example.com> SHOW CURRENT ROLES;
+-------+
| role  |
+-------+
+-------+
No rows selected (0.305 seconds)

# Verify that no roles yields no access
0: jdbc:hive2://server1.example.com:100> SHOW TABLES;
+----------+
| tab_name |
+----------+
+----------+
0: jdbc:hive2://server1.example.com:100> SELECT COUNT(*) FROM sample_07;
Error: Error while compiling statement:
FAILED: SemanticException No valid privileges (state=42000,code=40000)

# Set the current role to the hive_admin role
0: jdbc:hive2://server1.example.com:100> SET ROLE hive_admin;
No rows affected (0.176 seconds)

# Show list of current roles
0: jdbc:hive2://server1.example.com:100> SHOW CURRENT ROLES;
+-------------+
|    role     |
+-------------+
| hive_admin  |
+-------------+
1 row selected (0.404 seconds)

# Execute commands that are permitted
0: jdbc:hive2://server1.example.com:100> SHOW TABLES;
+------------+
|  tab_name  |
+------------+
| sample_07  |
| sample_08  |
```

```
+------------+
2 rows selected (0.536 seconds)
0: jdbc:hive2://server1.example.com:100> SELECT COUNT(*) FROM sample_07;
+-------+
| _c0   |
+-------+
| 823   |
+-------+
1 row selected (20.811 seconds)
```

> 使用 `WITH GRANT OPTION` 能大幅减轻全局 SQL 管理员的负担。通用的做法是，创建一个指定业务范围的 Hive 数据库，并将管理员权限分配给一个数据库专用的 admin 角色。这样指定业务范围后，就有一定灵活度管理它们拥有的数据的权限。要确定某角色是否有这个选项，可以用 `SHOW GRANT ROLE role_name` 命令查看列的 `grant_option`。

示例 7-14 中，命令由 HiveServer2 的 `beeline` CLI 执行，但它们也可以使用 `impala-shell` 运行。这两个组件使用了同样的 Sentry 服务，因而具有同样的 Sentry 策略，所以从其中一个组件进行的修改会立刻在另一个组件中反映出来。Sentry 授权决策不会被独立的组件缓存，这是从安全的后果考虑的。

7.7.2 SQL策略文件

使用为 SQL 组件提供 Sentry 服务的 Sentry 部署方式时，策略管理是熟悉且简洁易懂的，而使用传统的基于策略文件的实现则并非如此。启用 Sentry 的组件需要拥有对策略文件的读权限。在 Hive 和 Impala 中使用策略文件时，可以通过使 hive 组包含 file 组，并且确保 `hive` 和 `impala` 用户都是该组成员的方式实现。策略文件本身既可以放在本地文件系统，也可以放在 HDFS 中。对于前者，文件需要存储做出授权决策的组件被部署的地方。例如，Hive 启用 Sentry 时，本地策略文件需要存储在运行有 HiveServer2 守护进程的机器上。

> 为了利用 HDFS 同步复制机制带来的冗余和高可用性，强烈推荐在 HDFS 中指定一个位置存储策略文件。因为每一个用户操作都需要读取策略文件，明智的做法是增加该文件的副本，使得读取它的组件能够从许多不同的节点进行检索。这可以通过 `hdfs`

dfs -setrep *N* /user/hive/sentry/sentry-provider.ini 实现，此处的 *N* 是所需的副本数。策略文件很小，所以将副本的数量设为集群中 DataNode 的数量是完全可行的。

策略文件的格式遵守典型的 ini 文件格式，带有被中括号标识的配置段和独立的键值对配置部分。示例 7-15 展示了适用于 Hive 和 Impala 的 Sentry 样例配置文件。

示例 **7-15**　SQL sentry-provider.ini

```
[databases]
product = hdfs://nameservice1/user/hive/sentry/product.ini

[groups]
admins = admin_role, tmp_access
analysts = analyst_role, tmp_access
developers = developer_role, tmp_access
etl = etl_role, tmp_access

[roles]
# uri accesses
tmp_access = server=server1->uri=hdfs://nameservice1/tmp

# default database accesses
analyst_role = server=server1->db=default->table=*->action=select
developer_role = server=server1->db=default
etl_role = server=server1->db=default->table=*->action=insert, \
           server=server1->db=default->table=*->action=select

# administrative role
admin_role = server=server1
```

示例 7-15 中的策略文件做了很多事，可能并不能一眼就看出它定义了什么。策略文件的第一段是数据库。本段列出了所有数据库，以及为了确保对它们的安全访问所使用的相应策略文件。每个数据库不会强制要求拥有隔离的配置文件。然而，将配置文件隔离能带来以下好处：

- 能够进行版本跟踪，以轻易区分哪个数据库受到策略变化的影响以及受影响的时间；
- 能以单个数据库的粒度进行授权管理控制；
- 某个数据库策略文件的错误配置不会影响主策略文件

sentry-provider.ini 或其他数据库策略文件；

- 能够通过改变 HDFS 中对策略文件的访问权轻松禁用对整个数据库的访问，而不用改动策略文件本身的内容。

策略文件的第二段是组。本段提供了组和它们被分配到的角色之间的映射，语法是 group=role。正如之前讨论过的，组来自两个地方：来源于 Hadoop 的组，或者是本地专门为 Sentry 配置的组。示例 7-15 中没有定义本地配置的组，因为早先示例 7-1 中的 sentry-site.xml 被配置为 HadoopGroupResourceAuthorizationProvider。以下是关于策略文件中组配置段的几个要素：

- 给定组能够分配给许多角色，用逗号隔开；
- 组配置段中的条目是从上向下读的，从而使得同组中的重复条目能够覆盖任何先前的条目；
- 组名都是全局的，无论它们是在本地定义的还是由 Hadoop 提供的；
- 角色名是本地的，分配给一个组的角色名仅适用于配置它的那个文件。

策略文件的最后一段是角色。这里定义安全策略的主要部分。角色配置的语法是 role=permission。配置的许可部分看起来有些奇怪，因为这部分也是 key=value 形式的语法，但通过每个键值对配置之间的箭头实现了粒度更精确的许可定义。这可以通过 admin_role 权限的定义证明。该角色被授予 **server** 级别的全部访问权限。下一粒度级别是数据库级，或 **db** 级。developer_role 的权限定义授予了访问默认数据库的全部权限。这之后的下一级别是 **table** 级。该例展示了策略文件的另一个特性——它支持使用通配符选项指代“任意”表。

通配符仅用于代表任意值。通过部分名称匹配引用数据库表时，它们不能用作引用的开头和结尾。这看似是对通配符实现的一种限制，但用户应牢记，这是在制定安全策略。使用通配符进行部分名称匹配有可能会意外地将特权授予非授权用户。想象一下这样的场景，宾夕法尼亚州的一组开发者用户被授权能够访问任何名称是 pa 的表，但稍后人力资源部的用户开始使用这个集群，并建立了一个名为 payroll 的表，该表包含所有公司雇员的工资单。现在，宾夕法尼亚的开发者

组无意中能够访问保密信息了。因此，在安全策略中使用通配符时，应当格外小心。

操作行为属于权限最高级粒度的角色定义的上下文内。Hive 和 Impala 的表的上下文中，仅支持 `select` 和 `insert` 操作。这些操作是完全互斥的。如果一个角色对一个表既打算进行 select 操作，也打算进行 insert 操作，那么需要获得这两种操作的许可。组的配置段中，多项权限能够被分配给一个角色。为了实现这个目的，可以简单地将它们用逗号隔开。为了增加可读性，可使用反斜杠进行权限定义的换行连接，如示例 7-15 中的 `etl_role` 所示。

角色段的最后一部分讨论了 URI 的概念。示例 7-15 展示了 `tmp_access` 角色的一个 URI 访问许可。该许可允许用户做两件事：为本处数据创建外部表，从其有权限访问的其他本地表导出数据。

> 默认的 URI 访问只能在 `sentry-provider.ini` 中被指定，而不能在每个数据库各自的策略文件中被指定。进行限制的原因是，单独的管理员会维护一个数据库策略文件，但不会对其他策略文件也进行维护管理。如果管理员能够在策略文件中定义 URI 访问控制，他们就可以允许自己或其他任何人访问 HDFS 中对 `hive` 用户通过外部表可读的任意位置。如果这种行为需要被重写，则应当在 HiveServer2 中对 Java 配置选项 `sentry.allow.uri.db.policyfile=true` 进行配置。该配置应该仅能在所有管理员对所有 Sentry 策略文件都具有拥有同样的访问权的情况下使用。

7.7.3 Solr策略文件

Hive 和 Impala 可以通过 SQL 语法使用 Sentry 服务和管理策略，而 Solr 还在使用策略文件的方法。策略文件的格式类似于 SQL 中相应的部分，只是有几处变化。不同于 SQL 组件那样在数据库和表上操作，Solr 授权是在集合上操作的。同样地，Solr 的特权不含 SELECT 和 INSERT，而使用 Query 和 Update 代替。Solr 特权可以被设为 All，通过星号（*）实现。

示例 7-16 展示了一个与 SQL 示例相似的设计。在组配置段，组被赋值为角色；在角色配置段，角色被赋值为特权。`analyst_role` 提

供了查询 customer_logs 集合的权限，etl_role 提供了更新它的权限，developer_role 提供了访问它的所有权限。最后，admin_role 拥有针对 admin 集合的所有权限。

示例 **7-16**　Solr sentry-provider.ini

```
[groups]
admins = admin_role
analysts = analyst_role
developers = developer_role
etl = etl_role

[roles]
analyst_role = collection=customer_logs->action=Query
developer_role = collection=customer_logs->action=*
etl_role = collection=customer_logs->action=Update

# administrative role
admin_role = collection=admin->action=*
```

应当指出，虽然 SQL 策略文件允许每个数据库拥有独立的策略文件，但 Solr 却不允许这样做。这意味着 Solr 策略管理员在修改策略时需要格外当心，因为就像 SQL 策略文件那样，一个语法错误会使整个策略文件失效，从而无意间导致所有数据库都拒绝访问。一个对付拼写错误和语法错误的好办法是，使用 config-tool 验证策略文件，这正是下一节要讲的。

7.7.4　策略文件的验证和校验

既然 Sentry 一开始被设计为使用纯文本格式的策略文件，那么管理员显然需要某种校验工具，在文件生效之前对其进行基本的合理性检查。Sentry 附带了一个名为 sentry 的二进制程序（这名称真是毫无新意），它为策略文件的实现提供了一个重要功能：config-tool 命令。该命令不仅允许管理员检查策略文件中的错误，还提供了一个针对给定用户的特权验证机制。示例 7-17 展示了对策略文件的校验，其中第一个策略文件没有错误，第二个有一处拼写错误（server 被误拼为 sever）。

示例 **7-17**　Sentry config-tool 校验

```
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-too
-s file:///etc/sentry/sentry-site.xml -i file:///etc/sentry/sentry-provide
Using hive-conf-dir /etc/hive/conf
Configuration:
```

```
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
No errors found in the policy file
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-too
-s file:///etc/sentry/sentry-site.xml -i file:///etc/sentry/sentry-provide
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider2.ini
Sentry server: server1
 *** Found configuration problems ***
ERROR: Error processing file file:/etc/sentry/sentry-provider2.ini
 No authorizable found for sever=server1
ERROR: Failed to process global policy file
 file:/etc/sentry/sentry-provider2.ini
Sentry tool reported Errors:
org.apache.sentry.core.common.SentryConfigurationException:
        at org.apache.sentry.provider.file.SimpleFileProviderBackend.
         validatePolicy(SimpleFileProviderBackend.java:198)
        at org.apache.sentry.policy.db.SimpleDBPolicyEngine.
         validatePolicy(SimpleDBPolicyEngine.java:87)
        at org.apache.sentry.provider.common.ResourceAuthorizationProvider
         validateResource(ResourceAuthorizationProvider.java:170)
        at org.apache.sentry.binding.hive.authz.SentryConfigTool.
         validatePolicy(SentryConfigTool.java:247)
        at org.apache.sentry.binding.hive.authz.
         SentryConfigTool$CommandImpl.run(SentryConfigTool.java:638)
        at org.apache.sentry.SentryMain.main(SentryMain.java:94)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.
         invoke(NativeMethodAccessorImpl.java:57)
        at sun.reflect.DelegatingMethodAccessorImpl.
         invoke(DelegatingMethodAccessorImpl.java:43)
        at java.lang.reflect.Method.invoke(Method.java:606)
        at org.apache.hadoop.util.RunJar.main(RunJar.java:212
[root@server1 ~]#
```

`config-tool` 提供的另一个强大功能是验证用户的特权。该功能可以通过列出指定用户的所有权限实现，也可以更具体地通过测试指定用户能否被授权执行某次查询实现。示例 7-18 展示了这些功能的使用。

示例 **7-18**　Sentry config-tool 验证

```
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-too
```

```
 -s file:///etc/sentry/sentry-site.xml \
 -i file:///etc/sentry/sentry-provider.ini -l -u bob
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
Available privileges for user bob:
        server=server1
        server=server1->uri=hdfs://server1.example.com:8020/tmp
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-too
 -s file:///etc/sentry/sentry-site.xml \
 -i file:///etc/sentry/sentry-provider.ini -l -u alice
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
Available privileges for user alice:
        *** No permissions available ***
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-too
 -s file:///etc/sentry/sentry-site.xml -i file:///etc/sentry/sentry-provic
 -u bob -e "select * from sample_08"
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
User bob has privileges to run the query
[root@server1 ~]# sentry --hive-config /etc/hive/conf --command config-too
 -s file:///etc/sentry/sentry-site.xml -i file:///etc/sentry/sentry-provic
 -u alice -e "select * from sample_08"
Using hive-conf-dir /etc/hive/conf
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:/etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
FAILED: SemanticException No valid privileges
*** Missing privileges for user alice:
         server=server1->db=default->table=sample_08->action=select
User alice does NOT have privileges to run the query
Sentry tool reported Errors: Compilation error: FAILED:
SemanticException No valid privileges
[root@server1 ~]#
```

7.7.5　从策略文件迁移

Sentry 服务被添加入项目时，还有一个有用的迁移工具也被同时加入。该工具使 6 管理员能够从已存在的文件向 Sentry 服务后端数据库导入策略。这缓解了需要为每一条规则导出 SQL 语法，并手动将它们添加进数据库的痛苦。该迁移工具是 `config-tool` 的功能增强，最后一节将进行讲解。示例 7-19 展示了该工具的用法。

示例 **7-19**　Sentry 策略导入工具

```
[root@server1 ~]# sentry --command config-tool --import \
-i file:///etc/sentry/sentry-provider.ini
Using hive-conf-dir /etc/hive/conf/
Configuration:
Sentry package jar: file:/var/lib/sentry/sentry-binding-hive-1.4.0.jar
Hive config: file:/etc/hive/conf/hive-site.xml
Sentry config: file:///etc/sentry/sentry-site.xml
Sentry Policy: file:///etc/sentry/sentry-provider.ini
Sentry server: server1
CREATE ROLE analyst_role;
GRANT ROLE analyst_role TO GROUP analysts;
# server=server1
GRANT SELECT ON DATABASE default TO ROLE analyst_role;
CREATE ROLE admin_role;
CREATE ROLE developer_role;
CREATE ROLE etl_role;
GRANT ROLE admin_role TO GROUP admins;
GRANT ALL ON SERVER server1 TO ROLE admin_role;
GRANT ROLE developer_role TO GROUP developers;
# server=server1
GRANT ALL ON DATABASE default TO ROLE developer_role;
GRANT ROLE etl_role TO GROUP etl;
# server=server1
GRANT INSERT ON DATABASE default TO ROLE etl_role;
# server=server1
GRANT SELECT ON DATABASE default TO ROLE etl_role;
[root@server1 ~]#
```

7.8　小结

从概念上讲，Sentry 常见易懂。然而从本章可以看出，知易行难。尽管 Sentry 是 Hadoop 生态系统中相对较新的组件之一，它依然很快成为 Hadoop 安全中不可分割的一部分。 Sentry 在很短的一段时间内发生了迅速的演变，预计其他生态系统组件也会将 Sentry 集成，以便通过统一的方式提供强有力的授权控制。

我们已经顺利结束了关于验证和授权的庞大主题，下面看看审计，并理解用户活动在集群中的意义。

第 8 章　审计

目前为止，我们已经介绍了如何恰当地标识和验证用户和服务的身份，也阐述了怎样进行授权控制以限制用户和服务在集群中的行为。虽然这些各式各样的控制对定义和强化 Hadoop 集群安全模型起到了很好的作用，但仍然缺乏安全模型的一个关键组件：审计。审计又称审核，是追踪集群中用户和服务行为的机制。这是安全问题中的一个关键部分。因为如果没有审计，那么任何人都可能察觉不到安全被破坏。审计功能提供对发生事情的记录以完善安全模型，可以被用在以下几个方面。

主动审计

　　这种审计与其他警报机制一起使用。例如，若一个用户试图访问集群中的资源并被拒，主动审计能够生成一封邮件发送给安全管理员，警告他们这件事情的发生。

被动审计

　　被动审计是指那些不会产生某些警报的审计。这通常是业务的最低要求，因此，指定的审计员和安全管理员可以通过查询审计寻找某些事件。例如，假如集群中存在一个安全漏洞，安全管理员能够通过查询审计日志找到漏洞被利用期间被访问的那些数据。

安全合规

　　一个业务可能需要被审计，从而使之满足内部合规性或法律合规性。这是最常见的情况：存储在 HDFS 中的数据包含敏感信息，比如个人身份信息（PII）、信用卡号和银行账号之类的财务信息、一些敏感的商业信息（工资单和工资财务）等。

Hadoop 的组件处理审计的方法各有不同，这取决于组件的功能。HDFS 和 HBase 这类组件是数据存储系统，因此它们的可审计事件集中于读、写和访问数据。与之相反，MapReduce、Hive 和 Impala 这类组件是查询引擎和处理框架，因此可审计事件集中于终端用户的查询和作业。后面几小节将深入介绍每个组件，并从审计的角度阐述与组件的典型交互。

8.1 HDFS审计日志

HDFS 提供两种不同的审计日志，以达到两种不同的目的。第一种日志是 `hdfs-audit.log`，用于对一般用户活动进行审计，例如用户创建新文件、改变文件权限、请求目录列表等。第二种日志是 `SecurityAuth-hdfs.audit`，用于审计服务层面的授权活动。这些日志文件的建立涉及对 `log4j.category.SecurityLogger` 以及 `log4j.additivity.org.apache.hadoop.hdfs.server.namenode.` 的钩取。示例 8-1 展示了此过程。

示例 **8-1** HDFS log4j.properties

```
# other logging settings omitted
hdfs.audit.logger=${log.threshold},RFAAUDIT
hdfs.audit.log.maxfilesize=256MB
hdfs.audit.log.maxbackupindex=20
log4j.logger.org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit=
  ${hdfs.audit.logger}
log4j.additivity.org.apache.hadoop.hdfs.server.namenode.FSNamesystem.audit
log4j.appender.RFAAUDIT=org.apache.log4j.RollingFileAppender
log4j.appender.RFAAUDIT.File=${log.dir}/hdfs-audit.log
log4j.appender.RFAAUDIT.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAAUDIT.layout.ConversionPattern=%d{ISO8601} %p %c{2}: %m%
log4j.appender.RFAAUDIT.MaxFileSize=${hdfs.audit.log.maxfilesize}
log4j.appender.RFAAUDIT.MaxBackupIndex=${hdfs.audit.log.maxbackupindex}
hadoop.security.logger=INFO,RFAS
hadoop.security.log.maxfilesize=256MB
hadoop.security.log.maxbackupindex=20
log4j.category.SecurityLogger=${hadoop.security.logger}
log4j.additivity.SecurityLogger=false
hadoop.security.log.file=SecurityAuth-${user.name}.audit
log4j.appender.RFAS=org.apache.log4j.RollingFileAppender
log4j.appender.RFAS.File=${log.dir}/${hadoop.security.log.file}
log4j.appender.RFAS.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAS.layout.ConversionPattern=%d{ISO8601} %p %c: %m%n
log4j.appender.RFAS.MaxFileSize=${hadoop.security.log.maxfilesize}
log4j.appender.RFAS.MaxBackupIndex=${hadoop.security.log.maxbackupindex}
```

那么，一个可审计事件出现时，到底发生了什么呢？对于这部分示例，先进行以下假设：

- 用户 Alice 被 Kerberos 规则 `alice@EXAMPLE.COM` 识别出身份，并且顺利地使用 `kinit` 获得一个有效的票据授予票据（TGT）；

- Alice 在 HDFS 的主目录中进行了显示目录列表的操作；
- Alice 在 HDFS 主目录中创建了一个名为 `test` 的空文件；
- Alice 将该文件的权限更改为全部可写；
- Alice 试图将文件从主目录移至 `/user` 路径。

示例 8-2 中，Alice 进行了一些在 HDFS 中的典型操作。这些都是用户活动事件，所以检查 `hdfs-audit.log`，以查看 Alice 通过 HDFS 操作留下的痕迹（示例中的 log 文件已经处理以适合阅读）。

示例 **8-2**　hdfs-audit.log

```
...
2014-03-11 23:50:18,251 INFO FSNamesystem.audit: allowed=true ugi=alice@EX
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/alice dst=null per
2014-03-11 23:50:18,280 INFO FSNamesystem.audit: allowed=true ugi=alice@EX
 (auth:KERBEROS) ip=/10.1.1.1 cmd=listStatus src=/user/alice dst=null perm
2014-03-11 23:50:32,058 INFO FSNamesystem.audit: allowed=true ugi=alice@EX
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/alice/test dst=nul
2014-03-11 23:50:32,073 INFO FSNamesystem.audit: allowed=true ugi=alice@EX
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/alice dst=null per
2014-03-11 23:50:32,096 INFO FSNamesystem.audit: allowed=true ugi=alice@EX
 (auth:KERBEROS) ip=/10.1.1.1 cmd=create src=/user/alice/test dst=null
 perm=alice:alice:rw-r-----
2014-03-11 23:50:39,558 INFO FSNamesystem.audit: allowed=true ugi=alice@EX
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/alice/test dst=nul
2014-03-11 23:50:39,587 INFO FSNamesystem.audit: allowed=true ugi=alice@EX
 (auth:KERBEROS) ip=/10.1.1.1 cmd=setPermission src=/user/alice/test dst=n
 perm=alice:alice:rw-rw-rw-
2014-03-11 23:50:47,157 INFO FSNamesystem.audit: allowed=true ugi=alice@EX
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user dst=null perm=null
2014-03-11 23:50:47,185 INFO FSNamesystem.audit: allowed=true ugi=alice@EX
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/alice/test dst=nul
2014-03-11 23:50:47,187 INFO FSNamesystem.audit: allowed=true ugi=alice@EX
 (auth:KERBEROS) ip=/10.1.1.1 cmd=getfileinfo src=/user/test dst=null perm
2014-03-11 23:50:47,190 INFO FSNamesystem.audit: allowed=false ugi=alice@E
 (auth:KERBEROS) ip=/10.1.1.1 cmd=rename src=/user/alice/test dst=/user/te
...
```

如上所示，审计日志显示了 Alice 进行的每一项操作的相关信息。她进行的每一项操作首先都需要一个 `getfileinfo` 命令，接着是她所执行的多种操作内容（显示状态、创建、设置权限和重命名）。本日志中，产生事件的用户、事件发生的时间、执行操作的 IP 地址以及多种其他信息都很清楚。另外一个被记录的重要信息是，Alice 最

近一次试图进行的操作是将文件从她的主目录移出至一个她无权访问的位置，且该操作被拒绝。

8.2　MapReduce审计日志

MapReduce 审计与 HDFS 审计很相似，包括两个目的相近的审计日志。第一个日志文件 `mapredaudit.log` 用于审计诸如作业提交之类的用户活动。第二个日志文件 `SecurityAuth-mapred.audit` 用于审计服务级授权活动，就像 HDFS 日志那样。这些文件的 log4j 参数需要进行设定，用于设定这些参数的钩子是 `log4j.category.SecurityLogger` 和 `log4j.logger.org.apache.hadoop.mapred.AuditLogger`，如示例 8-3 所示。

示例 **8-3**　MapReduce log4j.properties

```
# other logging settings omitted
hadoop.security.logger=INFO,RFAS
hadoop.security.log.maxfilesize=256MB
hadoop.security.log.maxbackupindex=20
log4j.category.SecurityLogger=${hadoop.security.logger}
log4j.additivity.SecurityLogger=false
hadoop.security.log.file=SecurityAuth-${user.name}.audit
log4j.appender.RFAS=org.apache.log4j.RollingFileAppender
log4j.appender.RFAS.File=${log.dir}/${hadoop.security.log.file}
log4j.appender.RFAS.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAS.layout.ConversionPattern=%d{ISO8601} %p %c: %m%n
log4j.appender.RFAS.MaxFileSize=${hadoop.security.log.maxfilesize}
log4j.appender.RFAS.MaxBackupIndex=${hadoop.security.log.maxbackupindex}
mapred.audit.logger=${log.threshold},RFAAUDIT
mapred.audit.log.maxfilesize=256MB
mapred.audit.log.maxbackupindex=20
log4j.logger.org.apache.hadoop.mapred.AuditLogger=${mapred.audit.logger}
log4j.additivity.org.apache.hadoop.mapred.AuditLogger=false
log4j.appender.RFAAUDIT=org.apache.log4j.RollingFileAppender
log4j.appender.RFAAUDIT.File=${log.dir}/mapred-audit.log
log4j.appender.RFAAUDIT.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAAUDIT.layout.ConversionPattern=%d{ISO8601} %p %c{2}: %m%
log4j.appender.RFAAUDIT.MaxFileSize=${mapred.audit.log.maxfilesize}
log4j.appender.RFAAUDIT.MaxBackupIndex=${mapred.audit.log.maxbackupindex}
```

对于这个示例，进行以下假设：

- 用户 Bob 被 Kerberos 规则 `bob@EXAMPLE.COM` 识别出身份，并且已经成功地使用 `kinit` 获得一个有效的 TGT；

- 没有使用 MapReduce 服务级授权；
- Bob 提交了一个 MapReduce 作业；
- Bob 在 MapReduce 作业结束以前终止了它。日志中这一系列操作的结果如示例 8-4 和示例 8-5 所示。

示例 **8-4**　mapred-audit.log

```
...
2014-03-12 18:11:46,363 INFO mapred.AuditLogger: USER=bob IP=10.1.1.1
OPERATION=SUBMIT_JOB TARGET=job_201403112320_0001 RESULT=SUCCESS
...
```

示例 **8-5**　SecurityAuth-mapred.audit

```
...
2014-03-12 18:46:25,200 INFO SecurityLogger.org.apache.hadoop.ipc.Server:
 Auth successful for bob@EXAMPLE.COM (auth:SIMPLE)

2014-03-12 18:46:25,239 INFO SecurityLogger.org.apache.hadoop.security.
 authorize.ServiceAuthorizationManager: Authorization successful for
 bob@EXAMPLE.COM (auth:KERBEROS) for protocol=interface
org.apache.hadoop.mapred.JobSubmissionProtocol

2014-03-12 18:46:29,955 INFO SecurityLogger.org.apache.hadoop.ipc.Server:
 Auth successful for job_201403112320_0002 (auth:SIMPLE)

2014-03-12 18:46:29,976 INFO SecurityLogger.org.apache.hadoop.security.
 authorize.ServiceAuthorizationManager: Authorization successful for
 job_201403112320_0002 (auth:TOKEN) for protocol=interface
 org.apache.hadoop.mapred.TaskUmbilicalProtocol

...(more)...

2014-03-12 18:47:11,598 INFO SecurityLogger.org.apache.hadoop.ipc.Server:
 Auth successful for bob@EXAMPLE.COM (auth:SIMPLE)

2014-03-12 18:47:11,638 INFO SecurityLogger.org.apache.hadoop.security.
 authorize.ServiceAuthorizationManager: Authorization successful for
 bob@EXAMPLE.COM (auth:KERBEROS) for protocol=interface
 org.apache.hadoop.mapred.JobSubmissionProtocol
...
```

示例 8-4 很容易看懂：用户 Bob 执行了操作 `SUBMIT_JOB`，这导致 ID 为 `job_201403112320_0001` 的 MapReduce 作业的创建。正

如预期的那样，其他相关信息包括事件发生的时间日期以及 IP 地址。示例 8-5 中，情况有所不同。第一个条目显示 Bob 成功通过 JobTracker 的验证，第二个条目表明 Bob 已经被授权提交该作业。接下来的两个事件（为了简洁起见，其后的两个相同事件已被删去）显示了作业本身的操作。有趣的是，Bob 终止正在运行的作业时，出现的审计事件与其提交该作业时的审计事件一模一样。

8.3　YARN审计日志

YARN 审计日志事件穿插在守护进程的日志文件中，然而它们更容易辨识，因为它们的类名称会被事件记录。对于 Resource Manager，它的名称是 `org.apache.hadoop.yarn.server.resourcemanager.RMAuditLogger`，对于 Node Manager，则是 `org.apache.hadoop.yarn.server.nodemanager.NMAuditLogger`。可以使用这些类名从普通的应用程序日志中解析审计事件。 YARN 若要记录审计日志，需要对 `log4j` 参数进行设置。该操作的钩子是 `log4j.category.SecurityLogger`，示例 8-6 示范了如何进行设置。

示例 **8-6**　YARN log4j.properties

```
# other logging settings omitted
hadoop.security.logger=INFO,RFAS
hadoop.security.log.maxfilesize=256MB
hadoop.security.log.maxbackupindex=20
log4j.category.SecurityLogger=${hadoop.security.logger}
log4j.additivity.SecurityLogger=false
hadoop.security.log.file=SecurityAuth-${user.name}.audit
log4j.appender.RFAS=org.apache.log4j.RollingFileAppender
log4j.appender.RFAS.File=${log.dir}/${hadoop.security.log.file}
log4j.appender.RFAS.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAS.layout.ConversionPattern=%d{ISO8601} %p %c: %m%n
log4j.appender.RFAS.MaxFileSize=${hadoop.security.log.maxfilesize}
log4j.appender.RFAS.MaxBackupIndex=${hadoop.security.log.maxbackupindex}
```

本例中，用户 Alice 通过 YARN 提交了一个 MapReduce 作业，紧接着作业开始运行。示例 8-7 展示了针对 Resource Manager 的审计事件，示例 8-8 展示了 Node Manager 之一的审计事件。注意，这里为了简洁起见而省略了重复的审计类名，并且事件日志已经以适合阅读。

示例 8-7　YARN Resource Manager 审计事件

```
2014-12-27 12:49:35,182 INFO USER=alice IP=10.6.9.73
  OPERATION=Submit Application Request
  TARGET=ClientRMService RESULT=SUCCESS
  APPID=application_1419453547005_0001
2014-12-27 12:49:43,598 INFO USER=alice OPERATION=AM Allocated Container
  TARGET=SchedulerApp RESULT=SUCCESS APPID=application_1419453547005_0001
  CONTAINERID=container_1419453547005_0001_01_000001
2014-12-27 12:49:57,288 INFO USER=alice IP=10.6.9.75 OPERATION=Register Ap
  TARGET=ApplicationMasterService RESULT=SUCCESS APPID=application_141945
  APPATTEMPTID=appattempt_1419453547005_0001_000001
2014-12-27 12:50:02,375 INFO USER=alice OPERATION=AM Allocated Container
  TARGET=SchedulerApp RESULT=SUCCESS APPID=application_1419453547005_0001
  CONTAINERID=container_1419453547005_0001_01_000002
2014-12-27 12:50:02,376 INFO USER=alice OPERATION=AM Allocated Container
  TARGET=SchedulerApp RESULT=SUCCESS APPID=application_1419453547005_0001
  CONTAINERID=container_1419453547005_0001_01_000003
2014-12-27 12:50:19,361 INFO USER=alice OPERATION=AM Released Container
  TARGET=SchedulerApp RESULT=SUCCESS APPID=application_1419453547005_0001
  CONTAINERID=container_1419453547005_0001_01_000002
2014-12-27 12:50:21,436 INFO USER=alice OPERATION=AM Released Container
  TARGET=SchedulerApp RESULT=SUCCESS APPID=application_1419453547005_0001
  CONTAINERID=container_1419453547005_0001_01_000003
2014-12-27 12:50:27,954 INFO USER=alice OPERATION=AM Released Container
  TARGET=SchedulerApp RESULT=SUCCESS APPID=application_1419453547005_0001
  CONTAINERID=container_1419453547005_0001_01_000001
2014-12-27 12:50:27,963 INFO USER=alice OPERATION=Application Finished - S
  TARGET=RMAppManager RESULT=SUCCESS APPID=application_1419453547005_0001
```

示例 8-8　YARN Node Manager 审计事件

```
2014-12-27 12:49:43,956 INFO USER=alice IP=10.6.9.75
  OPERATION=Start Container Request TARGET=ContainerManageImpl RESULT=SUCC
  APPID=application_1419453547005_0001
  CONTAINERID=container_1419453547005_0001_01_000001
2014-12-27 12:50:27,105 INFO USER=alice OPERATION=Container Finished - Suc
  TARGET=ContainerImpl RESULT=SUCCESS APPID=application_1419453547005_0001
  CONTAINERID=container_1419453547005_0001_01_000001
2014-12-27 12:50:27,984 INFO USER=alice IP=10.6.9.75 OPERATION=Stop Contai
  TARGET=ContainerManageImpl RESULT=SUCCESS APPID=application_1419453547005
  CONTAINERID=container_1419453547005_0001_01_000001
```

YARN 的众多优点之一是，能够指定资源池。如前所述，资源池可以进行授权控制，从而使得只有特定的用户和组才能够提交给特定的资源池。接下来的示例中，Bob 试图向 Prod 资源池提交数据，但他没有权限。示例 8-9 展示了这种情况下的审计日志。再次强调，为了简

洁起见而省略了重复的审计类名，并且日志已经处理以适合阅读。

示例 **8-9**　YARN Resource Manager 审计事件

```
2014-12-27 13:56:35,886 INFO USER=bob IP=10.6.9.73
  OPERATION=Submit Application Request TARGET=ClientRMService
  RESULT=SUCCESS APPID=application_1419705820412_0002
2014-12-27 13:56:35,917 WARN USER=bob OPERATION=Application Finished - Fai
  TARGET=RMAppManager RESULT=FAILURE DESCRIPTION=App failed with state: FA
  PERMISSIONS=User bob cannot submit applications to queue root.prod
  APPID=application_1419705820412_0002
```

8.4　Hive审计日志

Hive 审计与 YARN 相似，没有专用的审计日志文件。审计事件实际发生在 Hive Metastore 服务日志内部，因此安全管理员从常规应用的日志事件获取相关审计信息会有些困难。然而同 YARN 一样，可以使用审计记录的类名辨识审计事件。其他 Hive 组件——如 HiveServer2——没有专用的审计，但仍然能从服务日志找到类似审计的信息。对于该例假设：

- 用户 Bob 被 Kerberos 规则 `bob@EXAMPLE.COM` 识别出身份，并且已经成功地使用 kinit 获得一个有效的 TGT；
- Bob 使用 `beeline` CLI 连接至 HiveServer2；
- Bob 首先执行 `show tables` 命令，以列出默认数据库中的表；
- 然后 Bob 执行 `select count(*) from sample_07`，以统计 `sample_07` 表中的记录数。这些操作的结果如示例 8-10 所示，该例已经调整了可读性。

示例 **8-10**　Hive Metastore 审计事件

```
...
2014-03-29 17:13:18,778 INFO org.apache.hadoop.hive.metastore.HiveMetaStor
 ugi=bob        ip=/10.1.1.1    cmd=get_database: default
...
2014-03-29 17:13:18,782 INFO org.apache.hadoop.hive.metastore.HiveMetaStor
 ugi=bob        ip=/10.1.1.1    cmd=get_tables: db=default pat=.*
...
2014-03-29 17:13:37,110 INFO org.apache.hadoop.hive.metastore.HiveMetaStor
```

```
ugi=bob        ip=/10.1.1.1    cmd=get_table : db=default tbl=sample_07
```

看看示例 8-10 中的审计事件显示的一些内容。首先，审计事件本身由 `org.apache.hadoop.hive.metastore.HiveMetaStore.audit` 标记，这使从日志搜索审计事件变得容易了一些。其次，这些审计事件在用户身份识别方面与之前见过的审计事件有着细微的区别。 Hive 仅显示用户名，而不是 Kerberos UPN（User Principal Name）全名。每个审计事件中，用户执行的操作是由 `cmd` 字段标识的。可以看到，`show tables` 的查询生成两个审计事件：`get_database` 和 `get_tables`。用于统计行数的实际 SQL 查询生成了一个审计事件，即 `get_table`。与其他组件中见过的审计事件一样，本审计事件也给出了执行操作的用户 IP 地址。

8.5　Cloudera Impala审计日志

Impala 审计事件被 Impala 守护进程（impalad）记录到专用的审计日志。审计日志目录位置由标识 `audit_event_log_dir` 指定，一个典型的选择是 `/var/log/impalad/audits`。这些日志文件到达由指定行数决定的一定大小后，将会滚动，文件大小由标识 `max_audit_event_log_file_size` 设定，通常合理的设定是 5000 行。对于 Impala 的例子，做出与 Hive 的例子一模一样的场景假设。这些操作的结果如示例 8-11 所示。

示例 **8-11**　Impala 守护进程的审计日志

```
....

{"1396114935263":{"query_id":"914b9eb1591546f0:ff4419eab4de439c",
 "session_id":"e643b5e102f653ec:94e0a3d4b3646ca3",
 "start_time":"2014-03-29 17:42:15.201945000","authorization_failure":fals
 "status":"","user":"bob","impersonator":null,"statement_type":"SHOW_TABLE
 "network_address":"::ffff:10.1.1.1:47569","sql_statement":"show tables",
 "catalog_objects":[]}}

{"1396115148996":{"query_id":"97443eddd3c172fd:34fe3f37c84d6ea8",
 "session_id":"e643b5e102f653ec:94e0a3d4b3646ca3",
 "start_time":"2014-03-29 17:45:48.850540000","authorization_failure":fals
 "status":"","user":"bob","impersonator":null,"statement_type":"QUERY",
 "network_address":"::ffff:10.1.1.1:47569","sql_statement":
 "select count(*) from sample_07","catalog_objects":
[{"name":"default.sample_07","object_type":"TABLE","privilege":"SELECT"}]
....
```

示例 8-11 显示出，Impala 的审计日志与其他 Hadoop 组件的日志在格式上截然不同。这些审计事件以 JSON 格式记录，因此可读性较差，但借助外部工具可以轻松使用这些数据。第一个审计事件展示了用户在 `statement_type` 字段中设定的操作类型，即 `SHOW_TABLES`。该信息也能用于 `sql_statement` 字段，以展示 Bob 进行的精确查询操作。第二个审计事件展示了查询的操作类型。

8.6　HBase审计日志

HBase 日志将审计事件记录到一个单独的日志文件，可以在相关的 `log4j.properties` 文件中对该日志文件进行配置。HBase 的架构是：客户端仅与负责执行指定操作的特定服务器进行交互，因此审计事件在整个 HBase 集群中都有分布。例如，创建、删除和修改表的操作是由 HBase Master 负责的；而类似扫描、插入和读取这种方法是指定给表中的某个特定区域的，所以由 RegionServer 记录这些事件。

HBase 对审计事件进行记录时，需要对 `log4j` 参数进行设置。用于进行该设置的钩子为 `log4j.logger.SecurityLogger`，示例 8-12 展示了如何进行设置。

示例 **8-12**　HBase log4j.properties

```
# other logging settings omitted
log4j.logger.SecurityLogger=TRACE, RFAS
log4j.additivity.SecurityLogger=false
log4j.appender.RFAS=org.apache.log4j.RollingFileAppender
log4j.appender.RFAS.File=${log.dir}/audit/SecurityAuth-hbase.audit
log4j.appender.RFAS.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAS.layout.ConversionPattern=%d{ISO8601} %p %c: %m%n
log4j.appender.RFAS.MaxFileSize=${max.log.file.size}
log4j.appender.RFAS.MaxBackupIndex=${max.log.file.backup.index}
```

本例进行了以下操作：

- HBase 超级用户创建了一个名为 `sample` 的表；
- HBase 超级用户授予用户 Alice 对 `sample` 表的 RW（读写）访问权限；

- HBase 超级用户授予用户 Bob 对 `sample` 表的 R（读）访问权限；
- Alice 试图创建一个名为 `sample2` 的新表，但被拒绝访问；
- Alice 向 `sample` 表插入一个值；
- Alice 扫描 `sample` 表；
- Bob 扫描 `sample` 表；
- Bob 试图向 `sample` 表插入一个值，但被拒绝访问。

HBase 审计能够缩小到一个特定的类，即 `SecurityLogger.org.apache.hadoop.hbase.security.access.A` 和其他组件中的日志事件一样。这个类贯穿于各类日志，但为了简洁起见，将它从示例 8-13 和 8-14 中略去。同样，为了保证可读性，这些示例的格式都经过了编辑。

示例 **8-13**　HBase master 的审计日志

```
2014-12-27 21:05:56,938 TRACE Access allowed for user hbase; reason:
  Global check allowed; remote address: /10.6.9.74; request: createTable;
  context: (user=hbase@EXAMPLE.COM, scope=sample, family=cf, action=CREATE
2014-12-27 21:06:09,484 TRACE Access allowed for user hbase; reason:
  Table permission granted; remote address: /10.6.9.74;
  request: getTableDescriptors; context: (user=hbase@EXAMPLE.COM,
  scope=sample, family=, action=ADMIN)
2014-12-27 21:06:16,620 TRACE Access allowed for user hbase; reason:
  Table permission granted; remote address: /10.6.9.74;
  request: getTableDescriptors; context: (user=hbase@EXAMPLE.COM,
  scope=sample, family=, action=ADMIN)
2014-12-27 21:07:02,102 TRACE Access denied for user alice; reason:
  Global check failed; remote address: /10.6.9.74; request:
  createTable; context: (user=alice@EXAMPLE.COM, scope=sample2,
  family=cf, action=CREATE)
```

由示例 8-13 可知，表创建的时间被清楚地记录下来。用户 `hbase` 被允许——而 Alice 被拒绝——创建一个表。此外，还可以从这个日志文件中看出，实际授予权限的行为并不明显。当该操作被记录为一个 `ADMIN` 操作，并且与这个表有同样的作用域，那么会没有任何标识表明该表权限被授予哪个用户。这是 HBase 的一个局限，很可能在将来的版本中得到改善。

示例 **8-14** HBase region server 审计日志

```
2014-12-27 21:07:15,411 TRACE Access allowed for user alice; reason:
  Table permission granted; remote address: /10.6.9.74; request: put;
  context: (user=alice@EXAMPLE.COM, scope=sample,
  family=cf:col1, action=WRITE)
2014-12-27 21:07:18,705 TRACE Access allowed for user alice; reason:
  Table permission granted; remote address: /10.6.9.74; request: scan;
  context: (user=alice@EXAMPLE.COM, scope=sample, family=cf, action=READ)
2014-12-27 21:07:47,263 TRACE Access allowed for user bob; reason:
  Table permission granted; remote address: /10.6.9.74; request: scan;
  context: (user=bob@EXAMPLE.COM, scope=sample, family=cf, action=READ)
2014-12-27 21:07:57,756 TRACE Access denied for user bob; reason:
  Failed qualifier check; remote address: /10.6.9.74; request: put;
  context: (user=bob@EXAMPLE.COM, scope=sample, family=cf:col1, action=WRI
```

示例 8-14 中，Alice 和 Bob 尝试进行的读和写操作被清楚地标识出来。它提供了表的相关信息、列族、列以及这项操作被允许或拒绝的原因。

8.7 Accumulo审计日志

与 HBase 相似，Accumulo 可以被配置为将审计事件记录到一个单独的日志文件。由于不要求 Accumulo 客户端在每次访问时都与一个单一、集中式的服务通信，因此审计日志分散于整个集群。例如，用户创建、删除或修改表时，操作会被 Accumulo Master 记录，而扫描和写入这类操作则会被实际处理操作的 TabletServer 记录。

默认的 Accumulo 配置模板是将审计记录关闭的。用户可以在 log4j 配置文件 `auditLog.xml` 中，将审计记录的日志级别设为 `INFO`，以启用记录功能。示例 8-15 展示了一个启用审计记录功能的示例文件 auditLog.xml。

示例 **8-15** Accumulo auditLog.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">

  <!-- Write out Audit info to an Audit file -->
  <appender name="Audit" class="org.apache.log4j.DailyRollingFileAppender">
    <param name="File"
     value="/var/log/accumulo/accumulo01.example.com.audit"/>
    <param name="MaxBackupIndex" value="10"/>
```

```
    <param name="DatePattern" value="'.'yyyy-MM-dd"/>
    <layout class="org.apache.log4j.PatternLayout">
      <param name="ConversionPattern"
       value="%d{yyyy-MM-dd HH:mm:ss,SSS/Z} [%c{2}] %-5p: %m%n"/>
    </layout>
  </appender>
  <logger name="Audit" additivity="false">
    <appender-ref ref="Audit" />
    <!-- Change level from OFF to INFO. default:<level value="OFF"/> -->
    <level value="INFO"/>
  </logger>

</log4j:configuration>
```

Accumulo 既审计普通用户的访问，也审计系统管理员的操作。每条审计记录包括操作状态（成功、失败、允许、拒绝）和进行该操作的用户，如果操作是远程请求，则还包括客户端地址。对于失败的请求，还会记录那些导致失败的异常。虽然个体的操作行为在细节上有所不同，但通常都会包含操作目标，以及诸如被访问的行和列范围之类的相关参数这种细节。表 8-1 展示了 Accumulo 记录的行为列表。

表**8-1**：**Accumulo**的审计操作

操作	描述
authenticate	用户通过 Accumulo 的身份验证
createUser	管理员创建一个新用户
dropUser	管理员删除用户
changePassword	管理员更换用户的密码
changeAuthorizations	管理员改变对用户的授权
grantSystemPermission	管理员将系统权限授予用户
grantTablePermission	管理员授予用户某表的权限
revokeSystemPermission	管理员撤销用户的系统授权
revokeTablePermission	管理员撤销用户对某表的权限
createTable	用户创建表
deleteTable	用户删除表
renameTable	用户对表进行重命名
cloneTable	用户复制表
scan	用户扫描一定范围的行内容
deleteData	用户删除表数据
bulkImport	用户发起批量数据导入
export	用户从一个集群向另一集群导出表
import	用户导入一张从别处导出的表

现在看看进行一些操作之后的审计日志。本例包含执行以下操作之后的审计结果：

- Accumulo 的 root 用户创建了名为 alice 的用户；
- Accumulo 的 root 用户创建了名为 bob 的用户；
- Accumulo 的 root 用户创建了名为 sample 的表；
- Accumulo 的 root 用户授予 alice 对 sample 表的 Table.READ 和 Table.WRITE 访问权限；
- Accumulo 的 root 用户授予 bob 对 sample 表的 Table.READ 访问权限；
- alice 试图创建新表 sample2，访问被拒绝；
- alice 向 sample 表插入一个值；
- alice 扫描了 sample 表；
- bob 扫描了 sample 表；
- bob 试图向 sample 表插入一个值，访问被拒绝。

示例 8-16 和示例 8-17 中的审计日志根据可读性进行了格式上的调整，但其他内容没有改动。

示例 **8-16**　Accumulo master 的审计日志

```
2014-12-27 16:40:11,673/-0800 [Audit] INFO : operation: permitted;
    user: root; action: createTable; targetTable: sample;
2014-12-27 16:40:28,563/-0800 [Audit] INFO : operation: denied;
    user: alice; action: createTable; targetTable: sample2;
```

示例 8-16 中可以看到，创建表的操作被清晰地记录下来。root 用户能被允许进行 createTable 操作，然而 alice 被拒绝。其他的管理员操作被记录在 TabletServer 日志中。

示例 **8-17**　Accumulo TabletServer 的审计日志

```
2014-12-27 16:39:49,262/-0800 [Audit] INFO : operation: success;
    user: root: action: createUser; targetUser: alice; Authorizations: ;
2014-12-27 16:40:02,226/-0800 [Audit] INFO : operation: success;
    user: root: action: createUser; targetUser: bob; Authorizations: ;
2014-12-27 16:40:13,226/-0800 [Audit] INFO : operation: success;
```

```
    user: root: action: grantTablePermission; permission: READ;
    targetTable: sample; targetUser: alice;
2014-12-27 16:40:13,292/-0800 [Audit] INFO : operation: success;
    user: root: action: grantTablePermission; permission: WRITE;
    targetTable: sample; targetUser: alice;
2014-12-27 16:40:13,442/-0800 [Audit] INFO : operation: success;
    user: root: action: grantTablePermission; permission: READ;
    targetTable: sample; targetUser: bob;
2014-12-27 16:40:30,529/-0800 [Audit] INFO : operation: permitted;
    user: alice; action: scan; targetTable: sample; authorizations: ;
    range: (-inf,+inf); columns: []; iterators: []; iteratorOptions: {};
2014-12-27 16:40:43,180/-0800 [Audit] INFO : operation: permitted;
    user: bob; action: scan; targetTable: sample; authorizations: ;
    range: (-inf,+inf); columns: []; iterators: []; iteratorOptions: {};
```

示例 8-17 展示了 `TabletServer` 审计日志的输出能看到 `createUser` 操作、被创建的用户和该用户被授予的权限，也可以看到经过授权的 `grantTablePermission` 操作、目标表和目标用户。最后还可以看到两个 scan 操作和查询的细节：行范围、列和所用迭代器。值得注意的是写操作的缺失，这是 Accumulo 审计框架当前的不足之处。同样，也看不到身份验证事件，因为这些事件是被 shell 自身记录的。

8.8 Sentry审计日志

第 7 章中，我们知道 Sentry 的最新版本使用一个服务处理授权请求，并管理与规则库的交互。审计过程中，对修改授权策略的事件进行审计是极其关键的。为了做到这一点， Sentry 需要被配置为捕获审计事件，而记录类 `sentry.hive.authorization.ddl.logger` 正是需要被配置的类之一。示例 8-18 展示了如何进行这种配置。

示例 **8-18** Sentry 服务的 og4j.properties

```
# other log settings omitted
log4j.logger.sentry.hive.authorization.ddl.logger=${sentry.audit.logger}
log4j.additivity.sentry.hive.authorization.ddl.logger=false
sentry.audit.logger=TRACE,RFAAUDIT
sentry.audit.log.maxfilesize=256MB
sentry.audit.log.maxbackupindex=20
log4j.appender.RFAAUDIT=org.apache.log4j.RollingFileAppender
log4j.appender.RFAAUDIT.File=${log.dir}/audit/sentry-audit.log
log4j.appender.RFAAUDIT.layout=org.apache.log4j.PatternLayout
log4j.appender.RFAAUDIT.layout.ConversionPattern=%d{ISO8601} %p %c{2}: %m%
log4j.appender.RFAAUDIT.MaxFileSize=${sentry.audit.log.maxfilesize}
```

```
log4j.appender.RFAAUDIT.MaxBackupIndex=${sentry.audit.log.maxbackupindex}
```

现在，Sentry 被配置为记录审计事件，下面看一个示例。该例中，Alice 是 Sentry 管理员， Bob 不是。Alice 使用 `beeline` 控制台创建了一个名为 `analyst` 的新角色，并将它分配给 `analystgrp` 组，然后授予它对默认数据库进行 `SELECT` 操作的特权。接下来，Bob 试图使用 `impala-shell` 创建一个新角色，但访问被拒绝。示例 8-19 展示了这些行为的记录。

示例 **8-19**　Sentry 服务审计日志

```
2015-01-02 11:17:10,753 INFO ddl.logger:
  {"serviceName":"Sentry-Service","userName":"alice","impersonator":
  "hive/server1.example.com@EXAMPLE.COM","ipAddress":"/10.6.9.74",
  "operation":"CREATE_ROLE","eventTime":"1420215430742","operationText":
  "CREATE ROLE analyst","allowed":"true","databaseName":null,
  "tableName":null,"resourcePath":null,"objectType":"ROLE"}
2015-01-02 11:17:37,537 INFO ddl.logger:
  {"serviceName":"Sentry-Service","userName":"alice","impersonator":
  "hive/server1.example.com@EXAMPLE.COM","ipAddress":"/10.6.9.74",
  "operation":"ADD_ROLE_TO_GROUP","eventTime":"1420215457536",
  "operationText":"GRANT ROLE analyst TO GROUP analystgrp","allowed":"true
  "databaseName":null,"tableName":null,"resourcePath":null,"objectType":"
2015-01-02 11:17:52,408 INFO ddl.logger:
  {"serviceName":"Sentry-Service","userName":"alice","impersonator":
  "hive/server1.example.com@EXAMPLE.COM","ipAddress":"/10.6.9.74",
  "operation":"GRANT_PRIVILEGE","eventTime":"1420215472407","operationText
  "GRANT SELECT ON DATABASE default TO ROLE analyst","allowed":"true",
  "databaseName":"default","tableName":"","resourcePath":"","objectType":"
2015-01-02 11:33:20,199 INFO ddl.logger:
  {"serviceName":"Sentry-Service","userName":"bob","impersonator":
  "impala/server1.example.com@EXAMPLE.COM","ipAddress":"/10.6.9.73",
  "operation":"CREATE_ROLE","eventTime":"1420216400199","operationText":
  "CREATE ROLE temp","allowed":"false","databaseName":null,"tableName":nul
  "resourcePath":null,"objectType":"ROLE"}
```

如日志所示，实际的审计记录是 JSON 格式的。该格式有利于外部的日志聚合和管理系统对日志进行处理，这在大型企业中是很重要的。

8.9　日志聚合

审计日志通常分散在集群中的许多节点（如果不是全部）。节点的数量乘以生成的独立日志文件的数量，使得掌握系统中正在发生的事情变成一项庞大的任务。强烈推荐的典型做法是，使用某种日志聚合系

统从集群中的所有节点拉取所有审计事件，放入一个集中化的位置进行存储和分析。当然，还有专门面向 Hadoop 的方法，以获取关于集群中发生事件的额外信息。即便如此，企业中已经部署的通用日志聚合系统也是管理 Hadoop 审计日志的好办法。

另一个有意思的日志聚合选择是，将其聚合后再导回 Hadoop 集群以进行分析。Hadoop 的安全用例是通用的，在 Hadoop 中分析审计事件也适合这些用例。如本章所示，审计事件通常都是结构化形式，以方便使用类似 Hive 或 Impala 的 SQL 工具进行查询。

8.10　小结

本章学习了 Hadoop 生态系统中的几种组件，并且阐述了用户与集群交互时被记录审计事件的类型。这些日志事件不仅对于审计普通用户在做什么很关键，对于发现非授权用户试图做什么也很关键。尽管 Hadoop 生态系统没有原生警告功能，但日志事件的架构有利于外部附加工具以一种更通用化的方式使用事件日志。主动警告是 Hadoop 生态系统正在开发的新功能，尽管如此，仍然有很多通用的日志聚合工具拥有对满足某种条件的事件进行警告的功能，这些工具在企业中很通用。

本章涉及的所有审计功能包含 AAA（认证、授权和审计）中的审计部分。之后将深入学习实际的数据——Hadoop 的生命线和大数据——是怎样受保护的。

第三部分　数据安全

第 9 章　数据保护

Eddie Garcia

目前为止，我们已经讲述了如何配置 Hadoop 以实施标准的 AAA 控制。本章将理解这些控制如何与第 1 章介绍的 CIA 准则一起，为数据保护提供基础。数据保护是一个宽泛的概念，包括从数据保密性到准用性等很多主题。其中需要特别关注的一个内容是加密。

加密是保护数据的一种常见方法。数据加密主要有两种类型：静态数据（**data-at-rest**）加密和动态数据（**data-in-transit**）加密，也称为线上（**over-the-wire**）加密。静态数据指的是，即使机器关闭后依旧可以存储的数据，包括硬盘、闪存盘、USB 记忆棒、内存卡、CD、DVD，甚至储藏箱中一些老式软盘或磁带中的数据。动态数据，顾名思义，是流动的数据，例如在互联网、USB 线、咖啡店Wi-Fi、手机基站或者从远程空间站到地球之间传输的数据。

9.1　加密算法

深入了解两种类型的数据加密之前，先简要介绍加密算法。加密算法定义了用于加密数据的数学方法。一种常见的加密算法是 **AES**（**Advanced Encryption Standard**，高级加密标准），它是由美国国家标准技术研究所（NIST）在 FIPS-197（http://1.usa.gov/1GFoldU）中建立的一个标准。

本书不会描述 AES 加密是如何工作的，推荐阅读 Christof Paar 和 Jan Palzl 编著的《深入浅出密码学：常用加密技术原理与应用》中的第 4 章。其他常见的加密算法还包括 DES、 RC4、Twofish 和 Blowfish。

加密数据时，密钥的大小很重要。通常来说，密钥越大，越难破解，但缺点是加密速度就越慢。处理很小的数据时，加密密钥的大小应该不超过数据本身。

使用 AES 时，所支持的常见密钥大小是 128 位、192 位和 256 位。当今的业界标准是 AES-256（256 位密钥）加密，但历史已经证明，这也是会变的。DES 和三重 DES（3 轮 DES 加密）一度是业界标准，但这两种加密算法都能被现在的计算机很轻易地暴力破解。

考虑到加密带来的性能开销，芯片制造商创造了硬件层函数以提升加密性能。这些增强可以使加密速度比软件加密提升多个数量级。一种流行的硬件加密技术是英特尔的 AES-NI。

对加密方法和算法有了基本了解之后，可以深入探讨全盘加密和文件系统加密。这些加密方法不需要特殊硬件，实现起来较为容易。

9.2　静态数据加密

假设 Alice 将一条给 Bob 的消息放在一个 USB 记忆棒中，并将其给了 Bob。但 Bob 在慌乱中不知道将这个 USB 记忆棒放到哪儿了，而 Eve 恰好捡到了它。出于好奇，Eve 将其连到 USB 上，尝试读取其中内容。幸运的是，Alice 对消息进行了加密，否则她就会陷入尴尬的局面。这个简单的例子中，加密保证了消息的机密性，除 Bob 之外，没有人知道解密数据的密码。Hadoop 生态系统的核心是 HDFS，它是很多其他组件的文件系统。直到最近， HDFS 本身才支持加密数据，这意味着需要采用其他加密方法。

> 这些年来，经常发生由笔记本电脑和手机丢失、硬盘处理不当、硬件被盗导致的敏感数据泄漏案件。静态数据加密能够减轻这些类型的数据泄漏，因为加密使得查看数据变得更加困难（但并非不可能）。

除了原生 HDFS 加密之外，下面看看另外三种选择。我们不会对每个方法深入探讨，因为一些方法是针对厂商的。这些方法在 HDFS 之下以透明的方式工作，因此不需要进行 Hadoop 相关配置。所有这些方法都会在某个磁盘从磁盘阵列中物理移除的情况下对数据进行保护。

加密磁盘

　　该方法与操作系统完全无关，HDFS 用于存储数据的物理磁盘，支持原生加密。不足之处在于，加密磁盘无法保护数据不被恶意用户和系统上运行的进程读取。

全盘加密

　　该方法通常在系统启动时工作，与加密磁盘方式不同的是，它不需要特殊的磁盘或硬件。这种技术存在若干种实现方案，通常根据操作系统的不同而不同。一些全盘加密方法支持操作系统根分区加密，一些只支持 HDFS 块所在的数据分区或数据卷加密。全盘加密也与加密磁盘方式有一样的不足，即无法保护数据不被恶意用户和系统上运

行的进程读取。

文件系统加密

该方法在操作系统层工作，不需要特殊的磁盘或硬件。这种技术存在若干种实现方案，根据操作系统的不同而不同。该方法不支持对根分区的文件系统加密，否则就变成“鸡生蛋还是蛋生鸡”的问题了：加密 OS 需要先启动，以对 OS 进行解密。文件系统加密的一个好处是，它能够保护数据不被恶意用户和系统上运行的进程窃取。如果一个加密的主目录使用一个某用户知道的密码进行加密，并且该用户自系统启动之后没有登录过，那么恶意用户或进程就不可能获得该密码解锁用户数据，即便是 root 用户也不行。

9.2.1 加密和密钥管理

Hadoop 生产集群通常有成千上万个节点，每个节点有 8~12 个磁盘，将静态数据加密扩展到 HDFS 之外的其他组件增大了潜在的复杂性。准备实施加密时，考虑如下几个额外的问题至关重要：

- 如何在加密情况下配置多个磁盘分区？
- 如何避免在系统启动时或在明文脚本中提供密码？

最后，大规模静态加密的难点在于密钥管理。下一节将要讨论的原生 HDFS 静态数据加密混合使用了两种方式：将加密密钥与文件元数据组合，以及依赖外部的 key server 管理密钥材料。我们讨论的其他静态数据加密技术一定程度上也需要密钥管理服务。

为密钥管理系统选择厂商很复杂，我们也无法为你的环境推荐某个厂商。但你选择时可以考虑如下几个重要标准：

- 解决方案是否支持硬件安全模块？
- 解决方案的扩展性如何（密钥个数以及每秒获取密钥的速度）？
- 解决方案是否支持 Hadoop 标准（如 KeyProvider 接口）？
- 管理成百上千个密钥的授权控制的难易程度如何？

如果正在使用预打包的 Hadoop 版本，那么寻找密钥管理系统厂商最简单的方法是，查找你的 Hadoop 厂商所认证的安全厂商。

9.2.2　HDFS静态数据加密

从 Hadoop 2.6 开始，HDFS 支持原生静态加密。这项功能并不被视为全盘加密或文件系统加密，而是另外一种名为应用层加密的类型。该方法中，数据被发送和到达存储位置之前，在应用层被加密。这种方法运行在操作系统层之上，不需要除 Hadoop 提供之外的任何特殊的操作系统软件包或硬件。要想了解更多关于原生 HDFS 加密设计的信息，可以阅读 HDFS 静态数据加密设计文档（http://bit.ly/1KZMeph）。

HDFS 中，需要加密的目录被分解成若干加密区（**encryption zone**），加密区中的每个文件都被使用唯一的数据加密密钥（**data encryption key**，**DEK**）进行加密，这就是加密区的区别所在。明文 DEK 不是永久的，而是用一个名为加密区密钥（**encryption zone key**，**EZK**）的区域级加密密钥，将 DEK 加密成加密 **DEK**（**encrypted DEK**，**EDEK**）。之后，EDEK 作为指定文件 NameNode 元数据中的扩展属性永久存在。

HDFS 加密区提供了一个能够镜像外部安全域的工具。以一个具有多部门的公司为例，每个部门都需要维护一些仅限本部门可用的数据集。可以通过在每个部门创建一个加密区以保护各部门的数据，从而省去了在认证密钥库（keystore）里为每个文件保存唯一密钥的开支。

如果 EDEK 存储在 HDFS 元数据中，那么 EZK 在哪里存储在哪里呢？这些密钥需要确保安全，因为泄露 EZK 会导致所有存储在该加密区的数据都能够被他人访问。为了防止 Hadoop 管理员访问 EZK 从而能够加密任何数据，EZK 不能存储在 HDFS 中。EZK 需要通过安全的 **key server** 来访问，key server 本身是一个用于处理 EZK 的存储和获取的独立软件。在较大的公司中，实际的存储组件由一个专用的硬件安全模块（**hardware security module**，**HSM**）来处理。通过该部署，key server 就成为了请求密钥的客户端和后端安全存储的软件接口。

为了进行职责分离，需要在 HDFS、HDFS 客户端和 key server 之间有个中间层。Hadoop 密钥管理服务（**Key Management Server**，**KMS**）的引入解决了这个问题。KMS 负责生成加密密钥（包括 EZK 和 DEK）、与 key server 通信以及解密 EDEK。KMS 通过名为 **KeyProvider** 的 Java API，与 key server 进行通信。稍后讲解 KeyProvider 的实现和配置。

为了更好地理解其工作流程，下面看看当一个 HDFS 客户端向 HDFS 加密区中写入一个新文件时，发生的事件序列。

(1) HDFS 客户端调用 `create()` 函数写新文件。

(2) NameNode 请求 KMS 使用给定的 EZK-id/ 版本创建一个 EDEK。

(3) KMS 生成一个新的 DEK。

(4)KMS 从 key server 获取 EZK。

(5)KMS 对 DEK 进行加密，形成 EDEK。

(6)KMS 将 EDEK 提交给 NameNode。

(7) NameNode 将 EDEK 保存为文件元数据的扩展属性。

(8)NameNode 将 EDEK 提交给 HDFS 客户端。

(9)HDFS 客户端提交 EDEK 到 KMS，并请求 DEK。

(10)KMS 向 key server 请求 EZK。

(11)KMS 使用 EZK 解密 EDEK。

(12)KMS 将 DEK 提交给 HDFS 客户端。

(13) HDFS 客户端使用 DEK 加密数据。

(14) HDFS 客户端向 HDFS 写入加密数据块。

读取加密文件的事件序列如下。

(1) HDFS 调用 `open()` 函数来读文件。

(2)NameNode 将 EDEK 提交给客户端。

(3)HDFS 客户端将 EDEK 和 EZK-id/ 版本传递给 KMS。

(4)KMS 向 key server 请求 EZK。

(5)KMS 使用 EZK 解密 EDEK。

(6)KMS 将 DEK 提交给 HDFS。

(7) HDFS 客户端读取加密数据块，并使用 DEK 对其解密。

读和写的事件中，都没有提到 HDFS 授权。授权检查在文件可以被创建或打开之前仍然是存在的。加密 / 解密步骤仅在 HDFS 授权检查之后进行。

> KMS 在 HDFS 加密中扮演着重要角色，因此不应被放在运行着其他 Hadoop 生态系统组件的服务器或者作为客户端边界节点的服务器上。需要有一种合适的职责分离，并且加密的关键操作和其他操作之间需要进行隔离。

由于 KMS 和 key server 以及 HDFS 客户端之间的通信都包含加密密钥的传递，因此使用 TLS（传输层安全协议）对该通信也进行加密绝对是至关重要的。下一节将讨论如何做到这一点。

01. 配置

本章介绍了很多关于 HDFS 加密的内容，但还没有讨论如何对这些内容进行配置。在每个 HDFS 节点和客户端节点的 `core-site.xml` 中设置如下参数。

`hadoop.security.key.provider.path`

KeyProvider 作为客户端，与加密密钥交互时使用的 URI。示例：`kms://https@kms.example.com:16000/kms`。

在 HDFS 服务端（NameNode 和 DataNode）侧，可以使用如下属性。

`dfs.encryption.key.provider.uri`

KeyProvider 与读写加密区使用的加密密钥交互时，使用的 URI。示例：`kms://https@kms.example.com:16000/kms`。

`hadoop.security.crypto.cipher.suite`

用于加密编码的加密套件。默认值：`AES/CTR/`

NoPadding

hadoop.security.crypto.codec.classes.aes.ctr.nopadd

由逗号分隔的、AES/CTR/NoPadding 密码编码器的实现方案。第一个实现方案有效时会使用第一个实现方案，其他为备用方案。默认值：org.apache.hadoop.crypto.OpensslAesCtrCryptoCodec,o

hadoop.security.crypto.jce.provider

JCE 提供者。默认值：None

hadoop.security.crypto.buffer.size

CryptoInputStream 和 CryptoOutputStream 使用的缓存大小。默认值：8192

如上所示，HDFS 的配置很少。在密码方面，HDFS 使用合适的默认值，因此要启用 HDFS 加密只需设置前两个配置，即 hadoop.security.key.provider.path 和 dfs.encryption.key.provider.uri。

要配置 Hadoop KMS，需要用到 kms-site.xml 这个配置文件。在 Hadoop KMS 节点上配置如下属性。

hadoop.kms.key.provider.uri

EZK 提供者的 URI。示例：jceks://file@/var/lib/kms/kms.keystore。

hadoop.kms.authentication.type

要使用的认证机制。示例：simple 或 Kerberos。

hadoop.kms.authentication.kerberos.keytab

服务认证所需的 Kerberos keytab 文件位置。

hadoop.kms.authentication.kerberos.principal

服务认证时应当使用的 SPN。示例：HTTP/kms.example.com@EXAMPLE.COM。

`hadoop.kms.authentication.kerberos.name.rules`

Kerberos 使用的 `auth_to_local` 规则。示例：`DEFAULT`。

`hadoop.kms.proxyuser.<user>.groups`

`<user>`（如 hdfs、hive、oozie）被允许模拟的组列表。

hadoop.kms.proxyuser.<user>__.hosts++

被允许进行身份模拟的 `<user>`（如 hdfs、hive、oozie）的来源主机列表。

> KMS 配置项中，有一个例子说明了其能够使用基于文件的 KeyProvider。那只是一个保存 EZK 的 Java keystore 文件。虽然这是建立和运行 HDFS 加密的一种快捷简单的方法，但只有在概念验证（POC）或开发环境中才推荐用这种方法进行测试。使用基于文件的 KeyProvider 会将 KMS 和 key server 功能放在同一个机器上，这没有提供要求的职责安全分离，也不具备实施额外的隔离控制的能力。此外，密钥的存储只是磁盘上的一个基础文件。如前所述，多数企业想要使用类似 KeyProvider 的独立服务，它为 EZK 使用更安全的存储方式，例如 HSM 提供的存储。

从 KMS 配置能够看出，可以使用 Kerberos 强认证，这绝对是推荐配置。由于 KMS 提供内容的敏感性，不应使用非 Kerberos 的部署。实际的 KMS 运行在 HTTP 协议之上，因此 KMS 客户端的 Kerberos 认证发生在 SPNEGO 上。所以，KMS 使用的 Kerberos 主体应当与 `HTTP/kms.example.com@EXAMPLE.COM` 类似——使用 HTTP 服务名。

上一节提到过，为 KMS 设置 TLS 线上加密是很重要的。要做到这一点，需要在 `kms-env.sh` 中为 KeyStore 和密码配置两个环境变量。KeyStore 文件仅是一个 .jks（Java KeyStore）文件，其位置由环境变量 `KMS_SSL_KEYSTORE_fiLE` 指定。如果 KeyStore 使用了密码保护（这也是应当做的！），则要在环

境变量 KMS_SSL_KEYSTORE_PASS 中指定密码。

02. **KMS授权**

与其他 Hadoop 组件一样，KMS 也能够使用访问控制表（ACL）限制对特定功能的访问。kms-acls.xml 文件存储了哪些用户和组可以执行哪些 KMS 功能的相关信息，如示例 9-1 所示。

示例 **9-1**　KMS 的 kms-acls.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
  <property>
    <name>hadoop.kms.blacklist.CREATE</name>
    <value>hdfs supergroup</value>
  </property>
  <property>
    <name>hadoop.kms.blacklist.DELETE</name>
    <value>hdfs supergroup</value>
  </property>
  <property>
    <name>hadoop.kms.blacklist.ROLLOVER</name>
    <value>hdfs supergroup</value>
  </property>
  <property>
    <name>hadoop.kms.blacklist.GET</name>
    <value>hdfs supergroup</value>
  </property>
  <property>
    <name>hadoop.kms.blacklist.GET_KEYS</name>
    <value>hdfs supergroup</value>
  </property>
  <property>
    <name>hadoop.kms.blacklist.SET_KEY_MATERIAL</name>
    <value>hdfs supergroup</value>
  </property>
  <property>
    <name>hadoop.kms.blacklist.DECRYPT_EEK</name>
    <value>hdfs supergroup</value>
  </property>
  <property>
    <name>default.key.acl.MANAGEMENT</name>
    <value> infosec</value>
  </property>
  <property>
    <name>default.key.acl.GENERATE_EEK</name>
    <value>hdfs supergroup</value>
  </property>
  <property>
```

```
    <name>default.key.acl.DECRYPT_EEK</name>
    <value> hadoopusers</value>
  </property>
  <property>
    <name>default.key.acl.READ</name>
    <value> infosec</value>
  </property>
</configuration>
```

示例 9-1 中的每个条目都有一个格式为“用户 1，用户 2 组 1，组 2”的值，与 6.2 节描述的一样。你可能注意到了黑名单的使用。为了将 Hadoop 管理员与实际数据强制隔离，Hadoop 管理员不应能够与 KMS 交互，也不能在 KMS 上执行操作。属于超级组（supergroup）的 Hadoop 管理员有遍历整个 HDFS 目录树的能力，而加密数据不应能被集群管理员解密，因此必须将这些用户加入黑名单。

记住，Hadoop KMS 是个通用的密钥管理服务，它所管理的密钥是没有含义的，或者说对它们的处理方式上是没有区别的。这意味着从 KMS 的角度看，EZK 和 DEK 是一样的。这就是为什么使用 KMS ACL 是很重要的。例如，默认情况下，CREATE 操作会返回实际的密钥内容。如果普通用户能够获取实际的 EZK 那就糟了，因为它可以用于解密 EDEK，从而获取整个加密区的内容。

> 允许任意用户创建密钥会带来一些潜在的安全风险。例如一个恶意用户可以轻易地写个脚本，不停创建新密钥，直至 KMS 和 / 或 key server 宕掉（如存储空间不足）。这实际上制造了一个拒绝服务（dos）的场景，导致所有加密数据都无法访问！使用限制性的 KMS ACL 可以授权一小组安全管理员能够创建和管理密钥。

建议（至少）建立 3 个不同的角色以应用 ACL：Hadoop 管理员、安全管理员和普通用户。该模型下，Hadoop 管理员只需能够请求 KMS 生成新 EDEK；安全管理员负责创建和维持 EZK；最后，集群的普通用户只能请求 KMS 解密给定的 EDEK。

继续研究这个模型，示例 9-1 展示了 Hadoop 管理员——hdfs 用户和 supergroup 组——被列入黑名单，从而被禁止执行

不必要的操作。此外，infosec 组是唯一一个被允许执行 MANAGEMENT 和 READ 功能的组。最后，hadoopusers 组只被允许执行 DECRYPT_EEK 功能。

示例 9-1 展示的是使用 default.key.acl 前缀标记的默认 ACL，也可以将 ACL 定义为特定名称的密钥，例如 key.acl.foo.READ，其中 foo 是密钥名称。下一节介绍 HDFS 加密客户端操作时，将讨论如何使用密钥名称。

03. 客户端操作

之前从工作流和配置的角度介绍了 HDFS 加密，现在需要了解把这一切都建立起来的客户端操作。首先看新 EZK 的创建。要做到这一点，需要使用 hadoop key 命令，该命令会输出创建的密钥，以及执行该请求的 KMS 的详细信息。

```
[bob@server1 ~]$ hadoop key -create myzonekey
myzonekey has been successfully created with options
 Options{cipher='AES/CTR/NoPadding', bitLength=128, description='nul
ibutes=null}.
KMSClientProvider[https://kms.example.com:16000/kms/v1/] has been up
[bob@server1 ~]$
```

现在可以在 HDFS 中创建一个新的加密区。使用 hdfs crypto 命令实现：

```
[bob@server1 ~]$ hdfs dfs -mkdir /myzone
[bob@server1 ~]$ hdfs crypto -createZone -keyName myzonekey -path /m
[bob@server1 ~]$ hdfs crypto -listZones
/myzone   myzonekey
[bob@server1 ~]$
```

从这开始，HDFS 客户端可以读写 /myzone 目录的文件，并对其进行透明加密 / 解密。

> HDFS 加密区的创建需要一个空目录，而不能是一个包含数据的目录。要对已经存在于 HDFS 中的数据加密，要将需要加密的目录重命名为一个临时名称，重新创建改目录，设置加密区，然后将数据复制回加密区。记住，原始数据之前是以未加密的形式存储在磁盘上

的，将数据加密不会删除磁盘上原本暴露的未加密敏感数据。

9.2.3　MapReduce2中间数据加密

HDFS 加密启用时，对数据的临时、中间版本进行保护也很重要。虽然对 MapReduce 作业的中间数据进行加密是可行的，但仍有一些注意事项：

- 中间数据加密是基于单个作业的（客户端配置）；
- 用户可能不知道源数据来自加密区；
    - 用户可能没有正确启用中间数据加密
    - 用户可能由于性能影响禁用了中间数据加密
- 中间数据加密仅支持 MR2，不支持 MR1。

表 9-1 中的作业配置属性用于启用中间数据加密。

表**9-1**：中间数据加密属性

属性	描述
mapreduce.job.encrypted-intermediate-data	设置为 true 以启用（默认值 false）
mapreduce.job.encrypted-intermediate-data-key-size-bits	加密密钥长度（默认：128）
mapreduce.job.encrypted-intermediate-data.buffer.kb	以 KB 为单位的缓存大小（默认值 128）

有实际需要时，开启和强制使用中间数据加密当然是理想的。我们希望后续的 Hadoop 发布版能够实现这项功能，从而使得 MapReduce 任务检测到自己正在读取来自加密区的数据时，能够自动加密中间文件，并且该特性无法被客户端重写。

9.2.4　Impala磁盘溢出加密

Impala 能够在内存无法容纳正在处理的所有数据时，对溢出到磁盘的数据进行加密。如果没有磁盘溢出加密，那么敏感数据有可能会在未加密的情况下被写回磁盘，这会对敏感数据造成危害，使在一开始对数据进行加密的优势毁于一旦。

为了让 Impala 守护进程保护溢出到磁盘的数据，需要配置如下启动

标识。

`disk_spill_encryption`

将该值设为 `true`，开启对队列中溢出到磁盘的所有数据的加密。默认值：`false`。数据即将溢出到磁盘时，它会被使用随机生成的 AES 256 位密钥进行加密；数据被从磁盘中读回时，随之解密。

`disk_spill_integrity`

将该值设为 `true`，开启对队列中溢出到磁盘的所有数据的完整性检查。默认值：`false`。数据即将溢出到磁盘时，该数据的 SHA256 散列值会被记录下来；从磁盘读回时，会再次计算其 SHA256 值，并与原来的值进行比较。这能够防止溢出到磁盘上的数据被篡改。

9.2.5　全盘加密

如果使用不支持本地加密的 HDFS 版本，或者需要对其他 Hadoop 生态系统组件使用的数据进行加密，那么可以考虑全盘加密或文件系统加密。先看一下使用 **Linux** 统一密钥设置（**Linux Unified Key Setup**，**LUKS**）的全盘加密。除 LUKS 之外，还有一些用于全盘加密的其他产品。我们将重点介绍 LUKS，因为它是在 Linux 上启用全盘加密的开源工具。

> 生产或真实数据中，不能轻易进行数据加密的试验，因为一个错误可能就会导致数据永久不可恢复。

大多数 LUKS 的实现使用 Linux 发行版中自带的 `cryptsetup` 和 `dm-crypt`：

- `cryptsetup` 提供创建、配置、管理加密卷的用户空间工具
- `dm-crypt` 提供加密块设备的 Linux 内核空间逻辑

示例 9-2 中，我们介绍了如何使用命令行在设备上配置 LUKS。一些 Linux 发行版具有的工具可以在操作系统安装时进行简单配置。建立存储磁盘、增加额外磁盘或对已有磁盘重新分区时，启用全盘加密可以像勾选一个复选框或选择一个选项这么简单。这隐藏了使用 `cryptsetup` 和 `dm-crypt` 的复杂过程。推荐尽可能使用发行版提供的工具。

在设备上建立 LUKS 时，数据会被覆盖。如果在已经有数据的设备上建立 LUKS，首先要对整个设备进行备份，配置完 LUKS 后再恢复数据。进行 LUKS 配置时要谨慎行事。

示例 **9-2**　LUKS 加密

(1)安装 crypsetup。

在 CentOS/RHEL 中：

```
[root@hadoop01 ~]# yum install cryptsetup-luks
```

On Debian/Ubuntu:

```
[root@hadoop01 ~]# apt-get install cryptsetup
```

(2) 建立 LUKS 存储设备。

```
[root@hadoop01 ~]# cryptsetup -y -v luksFormat /dev/xvdc
WARNING!
========
This will overwrite data on /dev/xvdc irrevocably.

Are you sure? (Type uppercase yes): YES
Enter LUKS passphrase:
Verify passphrase:
Command successful.
```

(3) 打开设备并将其映射到一个新设备。

```
[root@hadoop01 ~]# cryptsetup luksOpen /dev/xvdc data1
```

这在 /dev/mapper/data1 上创建了一个新的映射设备。

(4) 清除设备上的所有数据（这主要是清除数据头，但清除所有数据是个好的安全方法）。



```
[root@hadoop01 ~]# dd if=/dev/zero of=/dev/mapper/data1
```

前面的操作是在整个存储设备上写 0，因此，根据设备的大小和系统的速度，这会耗费几分钟到几小时。

(5) dd 命令完成后，可以创建文件系统；此处使用 ext4，但也可以使用 XFS 或其他想要的文件系统格式。

```
[root@hadoop01 ~]# mkfs.ext4 /dev/mapper/data1
```

(6) 既然有了一个带文件系统的加密设备，现在可以像普通文件系统一样加载它了。

```
[root@hadoop01 ~]# mkdir /data/dfs/data1
[root@hadoop01 ~]# mount /dev/mapper/data1 /data/dfs/data1
[root@hadoop01 ~]# df -H
[root@hadoop01 ~]# ls -l /data/dfs/data1
```

(7) 对于加载到 /data/dfs/data[2-*N*] 的其他设备，重复前面的步骤，然后使用 /data/dfs/data[1-*N*] 来为 HDFS 存储安装 Hadoop。

这只是一个示例，它没有涵盖如何提供开机密码、如何执行备份等很多其他方面。如果需要增加一个磁盘该怎么办？如果需要调整分区大小呢？在生产部署中，这些都是应该考虑的问题。

9.2.6 文件系统加密

很多产品提供文件系统加密，但我们主要关注 eCryptfs，因为它是 Linux 中一个常见的开源文件系统加密方案。

eCryptfs 有两个主要部分：ecryptfs-utils 和 ecryptfs。

- ecryptfs-utils 提供创建、配置和管理加密目录的用户空间工具
- ecryptfs 提供将加密文件系统放到现有文件系统目录之上的 Linux 内核空间逻辑

示例 9-3 将介绍如何使用命令行配置 eCryptfs。一些 Linux 发行版具有的工具能够在操作系统安装时进行简单配置。建立存储磁盘、增加额外磁盘或对已有磁盘重新分区时，可以像勾选一个复选框或选择一个选项这么简单。这隐藏了使用 `ecryptfs-utils` 和 `ecryptfs` 的复杂过程。我们推荐尽可能使用发行版提供的工具。

示例 **9-3** eCryptfs 加密

(1)安装 eCryptfs-utils。

在 CentOS/RHEL 中：

```
[root@hadoop01 ~]# yum install ecryptfs-utils
```

在 Debian/Ubuntu 中：

```
[root@hadoop01 ~]# apt-get install ecryptfs-utils
```

(2) 将一个新的加密文件系统挂载到空的 HDFS 数据目录上。

```
[root@hadoop01 ~]# mount -t ecryptfs /data/dfs/data1 /data/dfs/data1
Select key type to use for newly created files:
1) passphrase
2) tspi
3) openssl
Selection: 1
Passphrase:

Select cipher:
1) aes: blocksize = 16; min keysize = 16; max keysize = 32 (not loaded)
2) blowfish: blocksize = 16; min keysize = 16; max keysize = 56 (not loade
3) des3_ede: blocksize = 8; min keysize = 24; max keysize = 24 (not loaded
4) twofish: blocksize = 16; min keysize = 16; max keysize = 32 (not loaded
5) cast6: blocksize = 16; min keysize = 16; max keysize = 32 (not loaded)
6) cast5: blocksize = 8; min keysize = 5; max keysize = 16 (not loaded)

Selection [aes]: aes
Select key bytes:
1) 16
2) 32
3) 24
Selection [16]: 32
Enable plaintext passthrough (y/n) [n]: n
Enable filename encryption (y/n) [n]: n
Attempting to mount with the following options:
```

```
ecryptfs_unlink_sigs
ecryptfs_key_bytes=32
ecryptfs_cipher=aes
ecryptfs_sig= 9808e34a098f3814
WARNING: Based on the contents of [/root/.ecryptfs/sig-cache.txt],
it looks like you have never mounted with this key
before. This could mean that you have typed your
passphrase wrong.
Would you like to proceed with the mount (yes/no)? : yes
Would you like to append sig [9808e34a098f3814] to
[/root/.ecryptfs/sig-cache.txt]
in order to avoid this warning in the future (yes/no)? : yes
Successfully appended new sig to user sig cache file
Mounted eCryptfs
```

mount 命令过程中，你会被要求输入以字节为单位的密钥大小。之前我们将期望的密钥大小设为 256 位，由于 1 字节包含 8 位，因此此处选择 32 位密钥。

(3) 对于加载到 `/data/dfs/data[2-N]` 的其他设备，重复前面的步骤，然后使用 `/data/dfs/data[1-N]` 为 HDFS 存储安装 Hadoop。

这个例子没有涵盖如何提供开机密码、如何进行备份或者忘记密码时怎么办等其他方面。这些是在大规模使用加密方案时需要额外考虑的

9.2.7 Hadoop中重要数据的安全考虑

如果在为 Hadoop 的静态数据配置加密，需要注意，敏感数据不仅存在于 HDFS 上，也存在于运行在 MySQL、PostgreSQL、SQLite、Oracle 或 Derby 的 shuffle、溢出填充、临时文件、日志文件、交换文件、索引和元数据库等其他地方。第 10 章将介绍 Hadoop 用于提供 HDFS 之外的数据集加密的一些地方。

9.3 动态数据加密

上一节描述了如何使用加密保护静态数据，但数据正在网络上传输时，该怎么办呢？先举一个抽象的例子。当 Alice 和 Bob 还是孩子的时候，他们喜欢在课堂上传纸条。由于他们不是总坐在一起，因此需要依赖其他孩子帮忙传递。他们怎么对付那些可能想在传走纸条之前读一下纸条内容的爱管闲事的小孩呢？或者更糟的是，如果被老师抓

到并且向全班同学读出纸条内容会怎样呢？

作为一个聪明的孩子，Alice 想出一个她自己的字母表，字母表中包括一一映射英语字母的符号。他们不用英语字母，而是用这个自定义的字母表写自己的消息。Alice 和 Bob 可以提前交换一份映射关系，甚至记下这个字母表（毕竟只有 26 个符号）。现在，当他们传出自己的纸条时，只有拥有那个字母表副本的人才能读懂内容。

这种方法简单有效，但不是绝对安全的。更复杂的方法可能包括使用多个字母表或随机映射的字母表，以及一个告诉接收者该使用哪个映射的密钥。这对于传纸条来说可能是小题大做，但设计动态数据加密系统时，这很重要。

9.3.1 传输层安全

传输层安全（**Transport Layer Security，TLS**）是一个用于动态数据加密的安全协议。 TLS 替代了安全套接字层（**Secure Socket Layer，SSL**），SSL 是动态数据加密的一个早期标准。TLS 的初次定义是在基于 SSL 3.0 协议的 RFC 2246 中，它是由 Paul Kocher 设计的。由于 TLS 和 SSL 拥有相同的历史，所以虽然这两个协议不同，但它们总被交替使用。使用相同的库实现 SSL 和 TLS 也很常见。例如，OpenSSL 库包含 SSL 2.0 和 SSL 3.0 的实现，也包含 TLS 1.0、1.1 和 1.2 的实现。

第 4 章是启用强认证的协议，而 SSL/TLS 协议保护数据在网络传输时的安全。虽然最常见的是，SSL/TLS 以 HTTPS 协议的形式与 Web 流量联系在一起，但它其实是可用于保护任意套接字连接的通用协议。这使得你可以创建一个加密管道，其他协议可以放在该管道之上。就像 Kerberos 客户端信赖 KDC 一样，使用 SSL/TLS 的客户端信赖中心的证书颁发机构（**certificate authority，CA**）。

以下是支撑 SSL/TLS 的一些基本概念。

私钥（**Private key**）

只有签名证书所有者知道的非对称加密密钥。

公钥（**Public key**）

公开的、可用于加密数据的非对称加密密钥，加密的数据只能被对应的私钥解密。

证书签名请求（**CSR**）

发送到证书颁发机构以申请特定身份的加密消息。

签名证书

发送 CSR 到 CA 的结果。签名证书包括与私钥一起生成的公钥的一个副本。使用自签名证书而不是 CA 签名的证书也可以，但只在测试和开发系统中才推荐这么用。

PKCS #12

包含私钥和签名证书的一种文件格式。

虽然此处不讨论 SSL/TLS 的很多技术细节，但接下来的基本工作流程示例会介绍一些应当了解的内容。

01. 生成新证书

    (1) 想要接受 SSL/TLS 连接的服务的管理员生成一个公私钥对。

    (2) 管理员之后生成一个 CSR 并发送到 CA。

    (3) CA 验证服务器 / 服务（有时候是企业实体）的身份，然后生产一个签名证书。

    (4) 服务的管理员之后可以安装签名证书。

02. **SSL/TLS**握手

    (1) Alice 连接到 Bob 服务，Bob 服务向 Alice 展示一个 SSL/TLS 证书。

    (2) Alice 在它信任的第三方链中查找签发 Bob 的证书的 CA 证书。

    (3) Alice 和 Bob 服务交换公钥，然后为当前会话协商一个新创建的对称密钥。

    (4) Alice 向 Bob 发送一条信息，该信息使用通过安全交换得来的对称密钥进行动态加密。

(5) 即使 Even 捕获到从 Alice 到 Bob 的消息数据包，她也无法对其解密，因为她没有对称密钥。

这种设计的一个众所周知的实现就是 RSA 密钥交换算法。该算法中，紧跟着私钥和公钥对生成之后的是公钥的安全交换，它允许通信双方发送只能被指定接收者解密的加密消息。

> RSA 的名字由 Ron Rivest、Adi Shamir 和 Leonard Adlemanwho 的姓氏而来，他们 1977 年在 MIT 的时候写了关于该算法的一篇论文。在本书中你可以看到，除了 Kerberos 之外，还有很多我们现在使用的安全技术都源于 MIT。

要想更深入地了解 SSL/TLS，推荐阅读 John Viega、Matt Messier 和 Pravir Chandra 编著的图书 *Network Security with OpenSSL*，以及 Scott Oaks 编著的 *Java Security*(Second Edition)。

9.3.2 Hadoop动态数据加密

Hadoop 有多种网络通信方式，包括 RPC、TCP/IP 和 HTTP。MapReduce、JobTracker、TaskTracker、NameNode 和 DataNode 的 API 客户端均使用 RPC 调用。HDFS 客户端使用 TCP/IP 套接字进行数据传输。HTTP 协议用于 MapReduce shuffle，很多 Web UI 的守护进程也使用 HTTP 协议。

这三种网络通信每种都有不同的动态加密方法。接下来探讨这几种加密方法的基础知识，第 10 章会介绍一个 Flume SSL/TLS 配置的详细示例，第 11 章和第 12 章还会介绍 Oozie、HBase、Impala 和 Hue 中 SSL/TLS 的使用。

01. **Hadoop RPC加密**

Hadoop 的 RPC 实现支持 SASL，SASL 除了支持身份认证外，还提供了可选的信息完整性和信息加密。Hadoop 使用 Java 的 SASL 实现，支持以下几种模式。

- `auth`：用于客户端和服务端之间的身份验证
- `auth-int`：用于身份验证和完整性
- `auth-conf`：用于身份验证、完整性和机密性

Hadoop 中的 RPC 保护在 `core-site.xml` 文件中的 `hadoop.rpc.protection` 属性中配置。该属性可以被设置成如下值。

- `authentication`：默认值，将 SASL 设置为 `auth` 模式，仅提供身份验证。
- `integrity`：将 SASL 设置为 `auth-int` 模式，在身份验证之外增加完整性验证。
- `privacy`：将 SASL 设置为 `auth-conf` 模式，增加加密以保证完全的机密性。

配置 Hadoop RPC 保护，需要在 `core-site.xml` 中按照如下所示进行设置。（记住，需要重启所有守护进程才能使该设置生效。）

```
<property>
  <name>hadoop.rpc.protection</name>
  <value>privacy</value>
</property>
```

08. **HDFS数据传输协议加密**

HDFS 数据从一个 DataNode 传输到另一个 DataNode 时，或者在 DataNode 和客户端之间传输时，会使用一个名为 **HDFS** 数据传输协议（**HDFS data transfer protocol**）的 TCP/IP 直连套接字。数据传输加密启用时，会使用 Hadoop RPC 协议交换数据传输协议中使用的加密密钥。

配置数据传输加密需要在 `hdfs-site.xml` 文件中，将 `dfs.encrypt.data.transfer` 设置为 `true`——只有在 DataNode 上才要求进行此更改。RPC 会用于交换加密密钥，因此要设置 `hadoop.rpc.protection` 配置项为 `privacy`，以保证启用 RPC 加密，如前所述。加密算法也需要被配置成使用 AES。通过下面的代码配置 AES 加密：

```
<property>
  <name>dfs.encrypt.data.transfer</name>
  <value>true</value>
</property>
<property>
```

```
  <name>dfs.encrypt.data.transfer.cipher.suites</name>
  <value>AES/CTR/NoPadding</value>
</property>
<property>
  <name> dfs.encrypt.data.transfer.cipher.key.bitlength</name>
  <value>256</value> <!-- can also be set to 128 or 192 -->
</property>
```

使用 `dfs.encrypt.data.transfer.cipher.suites` 设置 AES 加密是 2.6 版本加入的一个较新的 Hadoop 功能。对于之前的版本，可以设置 `dfs.encrypt.data.transfer.algorithm` 为 `3des`（默认值）或 `rc4`，以分别选择 triple-DES 或 RC4。

需要重启 DataNode 和 NameNode 守护进程，使设置生效。整个过程可以手动完成，Hadoop 发行版可能也会提供自动化的方法启用 HDFS 数据传输加密。

09. Hadoop HTTP加密

谈到 HTTP 加密，有一个众所周知的成熟方法——使用 **HTTPS** 加密传输中的数据。 HTTPS 是一个使用 SSL/TLS 的 HTTP 增强。虽然 HTTPS 是很标准化的，但在 Hadoop 中对它进行配置并非如此。一些 Hadoop 组件支持 HTTPS，但它们的配置步骤并不都一样。

你可能会回忆起我们描述基本的 SSL/TLS 概念时，提到需要一些额外的文件，例如私钥、证书和 PKCS #12 包。使用 Java 时，这些文件存储在 Java **keystore** 里。

一些 Hadoop 组件对于其他服务来说既是 HTTPS 服务端也是客户端。下面是一些例子：

- HDFS、MapReduce 和 YARN 守护进程既作为 SSL 服务端也作为 SSL 客户端
- HBase 守护进程只作为 SSL 服务端
- Oozie 守护进程只作为 SSL 服务端

- Hue 作为以上所有的 SSL 客户端

我们不深入讨论 HTTPS 配置，而重点关注 MapReduce 加密 shuffle 和加密 Web UI 的配置，作为配置其他组件的出发点。

14. 加密**shuffle**和加密**web Web UI**

MR1 和 MR2 都支持加密 shuffle。在 MR1 中，设置 `core-site.xml` 文件中的 `hadoop.ssl.enabled` 属性可以启用加密 shuffle 和加密 Web UI。在 MR2 中，设置 `hadoop.ssl.enabled` 属性只能启用加密 Web UI 的功能，设置 `mapred-site.xml` 文件中的 `mapreduce.shuffle.ssl.enabled` 属性能够启用加密 shuffle 的功能。

> 与 Kerberos 一样，配置 HTTPS 时一个很重要的事情是，要使用完整的主机名配置你的服务器；并配置 DNS，使其能够在集群中正确解析这些主机名。

在 MR1 和 MR2 中，设置 `core-site.xml` 中的 `ssl` 属性都能启用加密 Web UI。在 MR1 中，该设置还会启用加密 shuffle。

```
<property>
  <name>hadoop.ssl.enabled</name>
  <value>true</value>
  <final>true</final>
</property>
```

仅对于 MR2，要在 **mapred-site.xml** 中设置加密 shuffle SSL 属性。

```
<property>
  <name>mapreduce.shuffle.ssl.enabled</name>
  <value>true</value>
  <final>true</final>
</property>
```

作为一个可选项，也可以设置 `hadoop.ssl.hostname.verifier` 属性控制如何进行主机

名验证。可选的值如下所示。

DEFAULT

主机名必须与第一个 CN 或任意 SAN（subject-alt name）匹配。如果 CN 或某个 SAN 中存在通配符，则匹配所有子域名。

DEFAULT_AND_LOCALHOST

与 DEFAULT 模式的作用一样，添加 localhost 主机，localhost.localdomain、127.0.0.1 和 ::1 总会通过主机名验证。

STRICT

与 DEFAULT 模式的作用相似，但只在同级匹配通配符。例如 *.example.com 能够匹配 one.example.com，但不能匹配 two.one.example.com。

ALLOW_ALL

接受任意主机名。该模式不安全，应当仅在测试中使用。

例如要支持 default 加上 localhost 模式，则按如下所示进行配置：

```
<property>
  <name>hadoop.ssl.hostname.verifier</name>
  <value>DEFAULT_AND_LOCALHOST</value>
  <final>true</final>
</property>
```

还需要更新 ssl-server.xml 和 ssl-client.xml 文件，这些文件通常在 /etc/hadoop/conf 目录下。ssl-server.xml 里的设置如表 9-2 所示。

表9-2：ssl-server.xml的Keystore和Truststore设置

属性	默认值	描述
ssl.server.keystore.type	jks	keystore 文件类型
ssl.server.keystore.locat	NONE	keystore 文件路径；该文件应当为 mapred 用户所有，

		mapred 用户必须拥有对其的独占读权限（即 400 权限）
ssl.server.keystore.password	NONE	keystore 文件的密码
ssl.server.truststore.type	jks	truststore 文件类型
ssl.server.truststore.location	NONE	truststore 文件路径；该文件应当为 mapred 用户所有，mapred 用户拥有对其的独占读权限（即 400）
ssl.server.truststore.password	NONE	truststore 文件密码
ssl.server.truststore.reload.interval	10000	truststore 文件刷新间隔的毫秒数

一个完整配置的 **ssl-server.xml** 示例如下：

```
<configuration>
  <!-- Server keystore -->
  <property>
    <name>ssl.server.keystore.type</name>
    <value>jks</value>
  </property>
  <property>
    <name>ssl.server.keystore.location</name>
    <value>/etc/hadoop/ssl/server/hadoop01.example.com.jks</value>
  </property>
  <property>
    <name>ssl.server.keystore.password</name>
    <value>super-secret-squirrel</value>
  </property>

  <!-- Server truststore -->
  <property>
    <name>ssl.server.truststore.type</name>
    <value>jks</value>
  </property>
  <property>
    <name>ssl.server.truststore.location</name>
    <value>/etc/hadoop/ssl/server/truststore.jks</value>
  </property>
  <property>
    <name>ssl.server.truststore.password</name>
    <value>changeit</value>
  </property>
  <property>
    <name>ssl.server.truststore.reload.interval</name>
    <value>10000</value>
  </property>
</configuration>
```

ssl-client.xml 文件中的配置见表 9-3。

表9-3：ssl-client.xml中的keystore和truststore设置

属性	默认值	描述
ssl.client.keystore.type	jks	keystore 文件类型
ssl.client.keystore.location	NONE	keystore 文件路径，该文件应当为 mapred 用户所有，所有可以运行 MapReduce 作业的用户应当具有对其的读权限（即 444 权限）
ssl.client.keystore.password	NONE	keystore 文件密码
ssl.client.truststore.type	jks	truststore 文件类型
ssl.client.truststore.location	NONE	truststore 文件路径，该文件应当为 mapred 用户所有，所有可以运行 MapReduce 作业的用户应当具有对其的读权限（即 444 权限）
ssl.client.truststore.password	NONE	truststore 文件密码
ssl.client.truststore.reload.interval	10000	truststore 文件刷新间隔的毫秒数

一个完整配置的 **ssl-client.xml** 文件如下所示：

```
<configuration>
  <!-- Client keystore -->
  <property>
    <name>ssl.client.keystore.type</name>
    <value>jks</value>
  </property>
  <property>
    <name>ssl.client.keystore.location</name>
    <value>/etc/hadoop/ssl/client/hadoop01.example.com.jks</value>
  </property>
  <property>
    <name>ssl.client.keystore.password</name>
    <value>super-secret-squirrel</value>
  </property>

  <!-- Client truststore -->
  <property>
    <name>ssl.client.truststore.type</name>
    <value>jks</value>
  </property>
  <property>
    <name>ssl.client.truststore.location</name>
    <value>/etc/hadoop/ssl/client/truststore.jks</value>
  </property>
```

```
  <property>
    <name>ssl.client.truststore.password</name>
    <value>changeit</value>
  </property>
  <property>
    <name>ssl.client.truststore.reload.interval</name>
    <value>10000</value>
  </property>
</configuration>
```

配置 SSL/TLS 时，一定要确保禁用明文服务。若一个服务运行在 HTTP 80 端口，又配置了 HTTPS 并运行在 443 端口，那么一定要禁用运行在 80 端口上的 HTTP。为了更强的保护级别，可以配置一个防火墙（如 iptables 软件防火墙）禁用对 80 端口的访问。

设置 ssl-server.xml 和 ssl-client.xml 文件后，需要重启 MR1 中的 TaskTracker 和 MR2 中的 NodeManager，使改动生效。

9.4　数据销毁和删除

涉及数据安全时，如何删除数据是很重要的一方面。如果用户恰好要重复使用集群中先前被用于处理敏感数据的服务器，首先需要销毁这些敏感数据。如示例 9-2 所示，使用 dd 命令将 LUKS 分区清零。

使用 GNU shred 工具能够对数据进行更彻底的销毁。shred 工具会用随机模式对文件或设备进行覆写，从而更好地混淆之前写入的数据。用户可以向 shred 传递大量迭代式，默认的迭代次数为 3 次。旧版 DoD 5220.22-M 标准强制要求进行 7 次覆写才能安全清除敏感数据。最安全的覆写模式是 Gutmann 方法，该方法要求使用随机数据和特定数据结合的模式进行 35 次覆写。

在大容量磁盘上进行 35 次覆写是一项非常耗时的工作。假设硬盘可以恒定 100 MB/s 的速度写数据，想要完全覆写一块 2 TB 的磁盘 35 次，将会超过 200 小时，大概相当于 8.5 天。

处理一个拥有数以百计的机器和上千上万块磁盘的集群时，即使执行并行化清除，也是一项艰巨的任务。

DoD 标准在近年来已经发展到，认为对于特别敏感的数据，做再多的覆写操作也不够。这种情况下，只有消磁或物理销毁是可接受的。还有很重要的一点是，被损坏的磁盘不能被覆写，因此只能进行消磁或物理销毁。磁盘故障很常见且数据足够敏感的大集群里，应当有合适的物理销毁计划。许多设备能对磁盘进行物理销毁，还有一些公司提供此类现场服务，他们会将粉碎设备带至现场。

9.5 小结

本章讨论了如何使用加密技术保护数据不受用户和 Hadoop 集群管理员的非授权访问。我们对比了保护静态数据和动态数据的区别，阐述了 HDFS 最近添加的本地静态数据加密和适用于早期版本的替代方法，以及用于 HDFS 外部静态数据加密的方法。还展示了数据处理或查询过程中生成的中间数据如何被加密，以进行端到端保护。

接下来，我们讨论了使用 Hadoop RPC 加密手段对动态数据进行保护的方法，继续通过学习 HDFS 数据传输协议加密的形式学习保护从 HDFS 客户端到 DataNode 以及多个 DataNode 之间的数据。我们还讨论了如何加密 HTTP 端点以及进行带 SSL/TLS 的 MapReduce shuffle。最后，介绍了永久销毁数据的方法，说明如何将对数据的保护延伸到硬件的整个使用周期。

接下来的两章将会分别探索将数据安全扩展至数据导入管道和客户端访问，从而全面保护 Hadoop 环境。

第 10 章　数据导入安全

前面的章节着重从存储和数据处理的角度讲解了 Hadoop 安全。我们假设你在 Hadoop 中已存有数据，且想要保护对这些数据的访问；或者你想要控制用户如何分享已有的分析资源，但还没有解释数据起初是如何被导入 Hadoop 的。

有许多种方法能够将数据导入 Hadoop。最简单的途径是使用 Hadoop 的 `put` 命令，从本地文件系统（如本地硬盘或挂载的 NFS）将文件复制到 HDFS，如示例 10-1 所示。

示例 **10-1**　从命令行导入文件

```
[alice@hadoop01 ~]$ hdfs dfs -put /mnt/data/sea*.json /data/raw/sea_fire_9
```

这种方法适用于某些数据集，但更常见的做法是，从已存在的关系型系统导入数据，或设置面向事件——或日志——的数据流。对于这些用例，用户会分别使用 Sqoop 和 Flume。

Sqoop 被设计为可从关系型数据库拉取数据到 Hadoop，或从 Hadoop 将数据推送到远程数据库。这两种情况下，Sqoop 会运行一个 MapReduce 作业，执行实际的数据传输。Sqoop 默认使用 JDBC 驱动，在映射任务和数据库之间传输数据，这称为通用模式。通过该模式，使用 Sqoop 对新数据进行存储将变得更容易，唯一的要求是 JDBC 驱动必须可用。出于性能方面的原因，Sqoop 还支持使用厂商的定制工具和接口的连接器以优化数据传输。要使用这些优化手段，用户可通过配置 `--direct` 选项启用直接（**direct**）模式。例如，对 MySQL 启用直接模式时，Sqoop 会使用 `mysqldump` 和 `mysqlimport` 工具以更高效地从 MySQL 抽取数据，或将数据导入 MySQL。

Flume 是一个分布式服务，能有效收集、聚合以及移动大量事件数据，Flume 用户需要部署代理，这些代理是用于传输数据的 Java 进程。一个事件是流经 Flume 的数据的最小单元，事件拥有一个二进制（字节数组）的载荷，和一个可选的、由字符串键 / 值对定义的属性集。每一个代理都配置有源模块、阱模块和通道。源模块是能够使用外部数据源的组件，它既能从外部数据源拉取事件，也能接受从外部数据源推送来的事件。Flume 源会连接到一个通道，该通道使得阱模

块能够使用这些来自源模块的事件。这个通道是完全被动的，它从源模块接受数据并一直保存数据，直到阱模块使用掉数据为止。一个 Flume 阱能将事件传输至外部数据存储或进程。

Flume 包括使用 Avro RPC（远程过程调用）的 AvroSource（Avro 源）和 AvroSink（Avro 阱）以传输事件。为了构建复杂的、分布式的数据流，可对某一个 Flume 代理的 AvroSink 进行配置，使之向另一个 Flume 代理的 AvroSource 发送事件数据。虽然 Flume 能够支持多种源模块和阱模块，但实现 agent 间数据传输功能的主要还是 AvroSource 和 AcroSink。因此，接下来的讨论内容将限定在它们之间。Flume 的可靠性由通道的配置决定。对于内存中的数据流通道，速度的优先级高于稳定性，而位于磁盘上的通道则支持完全的可恢复性。图 10-1 展示了一个双层 Flume 数据流的内部组件，包括代理和它们之间的连接。

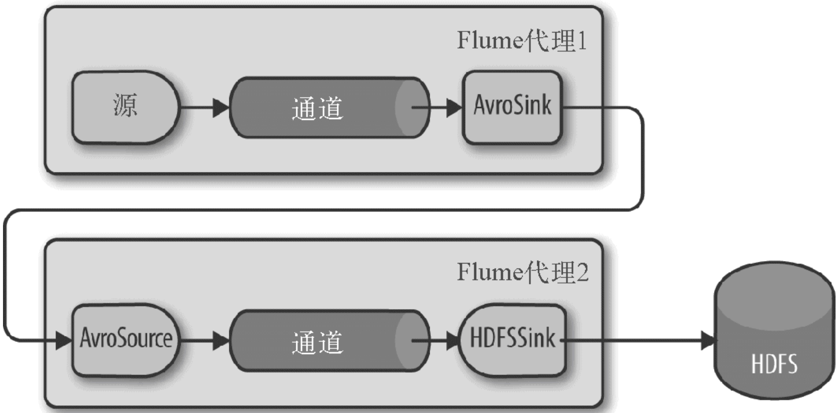


图 **10-1**：**Flume** 架构

因为 Sqoop 和 Flume 可能被用于传输敏感数据，所以考虑导入管道在整体部署中的安全性是很重要的，尤其需要关注导入管道的机密性、完整性和可用性（CIA）。机密性是指，将对数据的访问限制在一个拥有授权的用户集之中，通常系统会将身份验证、授权和加密结合在一起，以保证机密性的实现。完整性是指，用户能在多大程度上信任这些数据没有被篡改，大部分系统使用校验和或签名的方式验证数据完整性。可用性是指，保持信息资源的可使用性，在数据导入场景中，可用性意味着导入系统面对一些容量损失时具有鲁棒性。也就是说，系统处理一些下行系统或服务的突然中断时，具有保存传输数据的能力。

10.1　导入数据的完整性

分析过程的价值与数据的价值直接相关，而数据只有在可信时才具有价值，因此，数据的完整性是关键。Hadoop 是复杂的分布式系统，所以毫不意外地，数据导入流也经常具有同等的复杂性和分布性。这意味着数据能在许多地方被使用、损坏和篡改。你的具体威胁模型将决定导入管道所需的完整性的等级。大部分用例都会关注意外损坏的数据，而对于这些情况，对文件或记录的一个简单的校验和就足够了。为了防止数据被恶意用户篡改，可以添加加密签名和 / 或加密记录。

Flume 确保数据完整性的一个主要方法是，使用其内建的、可靠的通道实例。Flume 通道提供一个非常简单的接口，类似于一个无限长队列。通道使用 `put(Event event)` 方法将一个事件导入一个通道，使用 `take()` 方法将下一事件从通道取出。默认的通道实现是一个内存通道，当且仅当 Flume 的代理保持运行时，该实现才是可靠的。这意味着，进程或服务崩溃时，数据就会丢失。此外，由于事件始终不离开内存，Flume 会假定事件不被篡改，因而并不计算或验证事件的校验和。

对于那些关注数据的可靠传输和完整性的用户，Flume 提供了一种基于文件的通道。文件通道本质上实现了一个预写日志，每个事件被导入通道时，该日志被持久地存储到稳定的存储器。除了将事件存入磁盘之外，文件通道还为每个事件计算一个校验和，并将该校验和与事件一起写入预写日志。事件从通道中被取出时，它的校验和会被检验，以确保该事件没有被损坏。这提供了一定程度的完整性保障，然而这种保障对于经过通道传输的事件完整性来说仍然比较局限。目前，Flume 将事件从 AvroSink 的一个代理传递给 AvroSource 的另一个代理时，并不计算校验和。TCP 协议会保护数据包不受意外损坏，然而一个能够控制数据包的中间人仍能以不会被检测到的方式破坏数据。将了解到，Flume 没有能力加密 RPC 协议，而加密 RPC 恰好能够防止未被检测的中间人攻击。

继续讨论 Sqoop 提供的完整性之前，快速了解一下 Flume 如何实现可用性。Flume 允许用户创建一个能够保证“至少发送一次”消息处理语义的分布式数据流。严格地讲，只要第一个代理能够与外部的源通信，Flume 就能够与特定数据源进行通信。事件在代理间被依次处理时，任何下游代理的停机时间都可以通过使用一个故障切换阱处理器进行处理，该处理器以下游的两个或两个以上的 Flume 代理为目标。也可以使用负载均衡的阱处理器同时进行故障处理和负载均衡。这种

处理器将会把事件循环或随机地发送到下游阱节点的集合上，如果下游的阱节点出现故障，那么该处理器会尝试再下一个阱节点。

这两种机制都改善了整体数据流的可用性，但它们都不能保证任何某个特定事件的可用性。有人建议可增加一个复制通道，在告知源节点收到数据前，将事件复制到多个代理。但除非这些复制已经完成，否则事件在 Flume 节点宕机时仍会不可用。除非稳定存储文件的通道日志数据不可恢复，否则事件是不会丢失的。创建一个导入流时，记住这些注意事项很重要。我们将在 10.4 节进一步了解这些权衡措施。

与 Flume 不同，Sqoop 本身不支持验证导入数据的完整性。Sqoop 的优势在于，它可以利用同一个向 HDFS 写数据的进程从数据库拉取数据，这意味着，发生数据损坏的概率要比复杂的 Flume 流相对小一些。正如下一节将讲解的，可以通过开启 SSL 加密保证 Sqoop 的数据机密性。这同样也提高了破坏传输中数据的难度，从而改善数据完整性，然而并不能帮助验证写入 HDFS 的数据与数据库中数据是否一致。当前验证 Sqoop 导入的主流方法是往返验证表——从数据库到 Hadoop，再返回数据库。可以通过计算原始表的校验和与往返表的校验和验证数据没有丢失。遗憾的是，这个过程开销很高，并且可能需要进行多次全表扫描，具体扫描次数取决于数据库的校验性能。

10.2　数据导入的机密性

导入数据流的机密性等级依赖于导入的数据类型、关注的威胁模型和任何可能遵循的监管要求。数据被导入时，在网络中的传输过程以及在中间服务器的存储容易受到非授权用户的访问。甚至在一个受信任的合作方网络中，为了防止非授权用户有机会嗅探网络流量并获取敏感数据，某些数据必须始终被加密。同样地，数据传输所经服务器的管理员可能也没有权限访问某些被导入的数据。为了防止这些情况下的非授权访问，应当将数据在到达稳定的存储之前进行加密。

10.2.1　Flume加密

为了解决数据在网络传输过程中的非授权访问问题，Flume 支持在 AvroSource 和 AvroSink 上启用 SSL 加密。除了提供加密功能外，还可以给 AvroSource 和 AvroSink 配置信任策略，确保阱只将数据发往一个受信任的源。假设需要从一个运行在 `flume01.example.com` 上的 Flume 代理向运行在 `flume02.example.com` 上的代理发送事件，首先需要使用 `openssl` 命令行工具为 flume02 创建一个 RSA 私钥，如示例 10-2 所示。

示例 10-2　创建一个私钥

```
[alice@flume02 ~]$ mkdir certs
[alice@flume02 ~]$ cd certs
[alice@flume02 certs]$ openssl genrsa -des3 -out flume02.key 1024
Generating RSA private key, 1024 bit long modulus
...............................................................................
........++++++
.....................++++++
e is 65537 (0x10001)
Enter pass phrase for flume02.key:
Verifying - Enter pass phrase for flume02.key:
[alice@flume02 certs]$
```

示例 10-3 中，生成一个证书签名请求，因而会有一个证书被分发给刚刚创建的私钥。

示例 10-3　创建证书签名请求

```
[alice@flume02 certs]$ openssl req -new -key flume02.key -out flume02.csr
Enter pass phrase for flume02.key:
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
---
Country Name (2 letter code) [XX]:US
State or Province Name (full name) []:California
Locality Name (eg, city) [Default City]:San Francisco
Organization Name (eg, company) [Default Company Ltd]:Cluster, Inc.
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:flume02.example.c
Email Address []:admin@example.com

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
An optional company name []:
[alice@flume02 certs]$
```

一旦有了证书签名请求，就可以生成一个由受信任密钥签名的证书。示例中，没有根签名的权限，所以只生成一个自签名证书（给证书签名的密钥与请求证书的密钥相同）。实际部署中，应当将证书签名请求发送至公司的签名认证机构，他们会提供完成签名的证书。自签名

证书如示例 10-4 所示。

示例 10-4　创建一个自签名证书

```
[alice@flume02 certs]$ openssl x509 -req -days 365 -in flume02.csr \
  -signkey flume02.key -out flume02.crt
Signature ok
subject=/C=US/ST=California/L=San Francisco/O=Cluster, Inc./CN=flume02.clu
com/emailAddress=admin@example.com
Getting Private key
Enter pass phrase for flume02.key:
[alice@flume02 certs]$
```

我们将在 flume02 上使用刚刚创建的密钥和证书配置 AvroSink，但首先需要创建一个 truststore 供 flume01 验证密钥的真实性。实际部署中，truststore 可以由 CA（认证中心）或 Flume 集群的下属 CA 在进行签名时一起加载。这次将使用 Java 的 keytool，将证书导入 Java 的 truststore。

示例 10-5　创建一个 Java truststore

```
[alice@flume02 certs]$ keytool -import -alias flume02.example.com \
 -file flume02.crt -keystore flume.truststore
Enter keystore password:
Re-enter new password:
Owner: EMAILADDRESS=admin@example.com, CN=flume02.example.com, O="Cluster,
", L=San Francisco, ST=California, C=US
Issuer: EMAILADDRESS=admin@example.com, CN=flume02.example.com, O="Cluster
", L=San Francisco, ST=California, C=US
Serial number: 86a6cb314f86328b
Valid from: Tue Jun 24 11:31:50 PDT 2014 until: Wed Jun 24 11:31:50 PDT 20
Certificate fingerprints:
         MD5:  B6:4A:A7:98:9B:60:3F:A2:5E:0B:BA:BA:12:B4:8D:68
         SHA1: AB:F4:AB:B3:2D:E1:AF:71:28:8B:60:54:2D:C1:C9:A8:73:18:92:31
         SHA256: B1:DD:C9:1D:AD:57:FF:47:28:D9:7F:A8:A3:DF:9C:BE:30:C1:49:
95:AD:95:36:DC:40:4C:72:15:AB
         Signature algorithm name: SHA1withRSA
         Version: 1
Trust this certificate? [no]: yes
Certificate was added to keystore
[alice@flume02 certs]$
```

将认证和密钥用于 Flume 之前，需要将它们加载为一个 Java 可读的文件格式，通常会是 Java keystore 的 `.jks` 文件或 PCKS12 的 `.p12`。由于 Java 的 keytool 不支持密钥和证书的分开导入，使用

openssl 生成 PKCS12 文件并配置 Flume 直接使用该文件，如示例 10-6 所示。

示例 10-6　使用密钥和证书创建一个 PKCS12 文件

```
[alice@flume02 certs]$ openssl pkcs12 -export -in flume02.crt \
  -inkey flume02.key -out flume02.p12 -name flume02.example.com
Enter pass phrase for flume02.key:
Enter Export Password:
Verifying - Enter Export Password:
[alice@flume02 certs]$
```

配置 Flume 使用证书之前，需要将 PKCS12 文件移至 Flume 的配置目录，如示例 10-7 所示。

示例 10-7　复制 PKCS12 文件至 /etc/flume-ng/ssl 目录

```
[root@flume02 ~]# mkdir /etc/flume-ng/ssl
[root@flume02 ~]# cp ~alice/certs/flume02.p12 /etc/flume-ng/ssl
[root@flume02 ~]# chown -R root:flume /etc/flume-ng/ssl
[root@flume02 ~]# chmod 750 /etc/flume-ng/ssl
[root@flume02 ~]# chmod 640 /etc/flume-ng/ssl/flume02.p12
```

示例 10-8 中，需要将 truststore 复制到 flume01.example.com 上，这样阱会知道它能够信任位于 flume02.example.com 上的源。

示例 10-8　将 truststore 复制到 flume01.example.com

```
[root@flume02 ~]# scp ~alice/certs/flume.truststore flume01.example.com:/t
```

接下来，示例 10-9 中，将 truststore 转移至 Flume 的配置目录。

示例 10-9　转移 truststore 至 /etc/flume-ng/ssl

```
[root@flume01 ~]# mkdir /etc/flume-ng/ssl
[root@flume01 ~]# mv /tmp/flume.truststore /etc/flume-ng/ssl
[root@flume01 ~]# chown -R root:flume /etc/flume-ng/ssl
[root@flume01 ~]# chmod 750 /etc/flume-ng/ssl
[root@flume01 ~]# chmod 640 /etc/flume-ng/ssl/flume.truststore
```

现在，PKCS12 和 truststore 文件都已到位，可以配置 Flume 的源和阱，首先从 `flume01.example.com` 上的阱开始。关键配置参数如下所示。

`ssl`

启用 SSL 时设为 `true`。启用 SSL 时，还需要对 `trust-all-certs`、`truststore`、`truststore-password` 和 `truststore-type` 等参数进行配置。

`trust-all-certs`

要禁用证书验证时，将其设为 `true`。强烈推荐将该参数设为 `false`，因为这样能确保阱节点会检查与其连接的源确实正在使用一个受信任的证书。

`truststore`

将其设置为 Java truststore 文件的绝对路径。如果为空，Flume 将会使用默认的 Java 认证中心文件。Oracle JRE 装载有 `$JAVA_HOME/jre/lib/security/cacerts` 文件，Flume 将使用该文件，除非在 `$JAVA_HOME/jre/lib/security/jssecacerts` 创建一个指定位置的 truststore 文件。

`truststore-password`

将该参数设为保护 truststore 的口令。

`truststore-type`

将该参数设为 JKS 或另一种受支持的 truststore 类型。

具体配置如示例 10-10 所示。

示例 **10-10**　Avro SSL 阱的配置

```
a1.sinks = s1
a1.channels = c1
a1.sinks.s1.type = avro
a1.sinks.s1.channels = c1
a1.sinks.s1.hostname = flume02.example.com
a1.sinks.s1.port = 4141
```

```
a1.sinks.s1.ssl = true
a1.sinks.s1.trust-all-certs = false
a1.sinks.s1.truststore = /etc/flume-ng/ssl/flume.truststore
a1.sinks.s1.truststore-password = password
a1.sinks.s1.truststore-type = JKS
```

在 `flume02.example.com` 上，可以配置 AvroSource 配置，为使用证书和私钥来监听连接。密钥配置参数如下所示。

`ssl`

将其设为 true 以启用 SSL。启用 SSL 时，还需要对 `keystore`、`keystore-password` 和 `keystore-type` 参数进行配置。

`keystore`

将该参数设为 Java keystore 的绝对路径。

`keystore-password`

将该参数设为保护 keystore 的口令。

`keystore-type`

将其设为 `JKS` 或 `PKCS12`。

示例 **10-11**　Avro SSL 源的配置

```
a2.sources = r1
a2.channels = c1
a2.sources.r1.type = avro
a2.sources.r1.channels = c1
a2.sources.r1.bind = 0.0.0.0
a2.sources.r1.port = 4141
a2.sources.r1.ssl = true
a2.sources.r1.keystore = /etc/flume-ng/ssl/flume02.p12
a2.sources.r1.keystore-password = password
a2.sources.r1.keystore-type = PKCS12
```

除了保护传输中的数据，可能还需要确保数据在 Flume 传输通道中被写入磁盘时是加密的。一个方法是，在 Flume 通道中写数据的磁盘上使用支持全盘加密的第三方加密工具，也可以通过 dm-crypt/LUKS1

完成。然而，全盘加密也可能是过度保护，特别是当 Flume 不是使用该日志磁盘的的唯一服务，或不是所有事件都需要被加密时。

> 1为了使设置 dm-crypt/LUKS 更容易，可以使用 cryptsetup 工具。通过 cryptsetup 工具设置 dmcrypt/LUKS 的说明可以在 cryptsetup 的 FAQ 页面找到（http://bit.ly/1Hc9zCz）。

对于那些使用场景，Flume 提供了“仅加密被文件通道使用的日志文件”的功能。当前的实现只支持 Counter 模式下无填充的 AES 加密（AES/CTR/NOPADDING），但未来有可能再添加其他的加密算法和模式。目前，Flume 只支持 **JCE keystore** 的实现（**JCEKS**）作为密钥提供程序，但也不排除添加其他密钥支持程序。这需要对 Flume 本身进行修改，因为现在并没有能支持添加密钥提供程序的可插拔接口。除了这些限制，Flume 支持密钥轮换，帮助改进其安全性。因为文件通道的日志文件的生命周期相对较短，可以视需要尽可能频繁地轮换密钥以满足需求。为了确保用以前密钥写入的日志文件仍然可读，在最新的密钥被用于写入时，也必须保留旧的密钥用于读取文件。

要设置 Flume 的文件通道磁盘加密，先生成密钥，如示例 10-12 所示。

示例 **10-12**　为文件通道的磁盘加密生成密钥

```
[root@flume01 ~]# mkdir keys
[root@flume01 ~]# cd keys/
[root@flume01 keys]# keytool -genseckey -alias key-0 -keyalg AES -keysize
  -validity 9000 -keystore flume.keystore -storetype jceks
Enter keystore password:
Re-enter new password:
Enter key password for <key-0>
        (RETURN if same as keystore password):
Re-enter new password:
[root@flume01 keys]#
```

示例中，将 keystore 的口令设为 `keyStorePassword`，将密钥口令设为 `keyPassword`。然而在真实环境的部署中，应当使用更强的口令。keytool 不会显示正在输入的字符，也不会显示我们熟悉的星号字符，因此需谨慎输入口令。还可以分别通过命令行的 `-storepass keyStorePassword` 和 `-keypass keyPassword` 命令提供 keystore 口令和密钥口令。通常不推荐在命令行中写入口令，因为它们一般都会被写入 shell 的历史记录文件，而该文件不能

被视作安全的。接下来，将 keystore 复制到 Flume 的配置目录，如示例 10-13 所示。

示例 **10-13**　复制 keystore 到 Flume 的配置目录

```
[root@flume01 ~]# mkdir /etc/flume-ng/encryption
[root@flume01 ~]# cp ~/keys/flume.keystore /etc/flume-ng/encryption/
[root@flume01 ~]# cat > /etc/flume-ng/encryption/keystore.password
keyStorePassword
^D
[root@flume01 ~]# cat > /etc/flume-ng/encryption/key-0.password
keyPassword
^D
[root@flume01 ~]# chown -R root:flume /etc/flume-ng/encryption
[root@flume01 ~]# chmod 750 /etc/flume-ng/encryption
[root@flume01 ~]# chmod 640 /etc/flume-ng/encryption/*
[root@flume01 ~]#
```

要注意，还创建了包含 keystore 口令和密钥口令的文件。屏幕显示 `^D` 时，应当按住 Control 键并键入字母 D。生成这些文件有助于保护口令，因为它们不能被能够读取 Flume 配置文件的同一个用户所访问。现在配置 Flume 以启用文件通道加密，如示例 10-14 所示。

示例 **10-14**　加密文件通道的配置

```
a1.channels = c1
a1.channels.c1.type = file
a1.channels.c1.checkpointDir = /data/01/flume/checkpoint
a1.channels.c1.dataDirs = /data/02/flume/data,/data/03/flume/data
a1.channels.c1.encryption.cipherProvider = AESCTRNOPADDING
a1.channels.c1.encryption.activeKey = key-0
a1.channels.c1.encryption.keyProvider = JCEKSFILE
a1.channels.c1.encryption.keyProvider.keyStoreFile =
  /etc/flume-ng/encryption/flume.keystore
a1.channels.c1.encryption.keyProvider.keyStorePasswordFile =
  /etc/flume-ng/encryption/keystore.password
a1.channels.c1.encryption.keys = key-0
a1.channels.c1.encryption.keys.key-0.passwordFile =
  /etc/flume-ng/encryption/key-0.password
```



示例 10-14 至示例 10-16 展示了一些配置内容（`a1.channels.c1.encryption.keyProvider.keyStorePa`
`file`,
`a1.channels.c1.encryption.keys.key-0.passwordfile`）被分成两行。这是为了改进示例的可读性，但在

文件中是无效的。所有配置的参数名和值都必须位于同一行。

随着时间的推移，有必要轮换至新的加密密钥，以降低旧密钥遭到破解的风险。Flume 支持配置多个解密密钥，但仅使用最新的密钥用于加密。为了保证轮换密钥前写入的旧日志文件仍然可读，旧的密钥必须保留。示例 10-15 中，可以生成一个新密钥并更新 Flume，使之成为活动密钥。

示例 **10-15**　为磁盘上的文件通道加密生成一个新密钥

```
[root@flume01 ~]# keytool -genseckey -alias key-1 -keyalg AES -keysize 256 \
  -validity 9000 -keystore /etc/flume-ng/encryption/flume.keystore \
  -storetype jceks
Enter keystore password:
Enter key password for <key-1>
        (RETURN if same as keystore password):
Re-enter new password:
[root@flume01 ~]# cat > /etc/flume-ng/encryption/key-1.password
key1Password
^D
[root@flume01 ~]# chmod 640 /etc/flume-ng/encryption/*
[root@flume01 ~]#
```

现在，已经向 keystore 添加了新密钥，并且创建了相关的密钥口令文件，下面可以更新 Flume 的配置以使这个新密钥生效变为活动密钥，如示例 10-16 所示。

示例 **10-16**　加密文件通道的新密钥配置

```
a1.channels = c1
a1.channels.c1.type = file
a1.channels.c1.checkpointDir = /data/01/flume/checkpoint
a1.channels.c1.dataDirs = /data/02/flume/data,/data/03/flume/data
a1.channels.c1.encryption.cipherProvider = AESCTRNOPADDING
a1.channels.c1.encryption.activeKey = key-1
a1.channels.c1.encryption.keyProvider = JCEKSFILE
a1.channels.c1.encryption.keyProvider.keyStoreFile =
  /etc/flume-ng/encryption/flume.keystore
a1.channels.c1.encryption.keyProvider.keyStorePasswordFile =
  /etc/flume-ng/encryption/keystore.password
a1.channels.c1.encryption.keys = key-0 key-1
a1.channels.c1.encryption.keys.key-0.passwordFile =
  /etc/flume-ng/encryption/key-0.password
```

```
a1.channels.c1.encryption.keys.key-1.passwordFile =
  /etc/flume-ng/encryption/key-1.password
```

以下是文件通道加密的配置参数。

`encryption.activeKey`

　　用于加密新数据的密钥别名。

`encryption.cipherProvider`

　　密码算法提供程序的类型。支持的类型：`AESCTRNOPADDING`

`encryption.keyProvider`

　　密钥提供程序的类型。支持的类型：`JCEKSfiLE`

`encryption.keyProvider.keyStorefile`

　　keystore 文件的路径。

`encryption.keyProvider.keyStorePasswordfile`

　　含有 keystore 口令的文件的路径。

`encryption.keyProvider.keys`

　　有限长的曾用或现用的密钥别名列表。

`encryption.keyProvider.keys.<key>.passwordfile`

　　包含密钥 `key` 的口令文件的可选路径。如果忽略该参数，keystore 口令文件中的口令会被用于所有密钥。

10.2.2　Sqoop加密

与 Flume 不同，Sqoop 没有自己原生的对线上加密的支持。这并不奇怪，因为 Sqoop 依赖于标准的 JDBC 驱动和专门针对数据库优化的连接器。然而，Sqoop 可以被配置为连接到一个支持 SSL 的数据库并启用 SSL，这将启用对 JDBC 通道中数据的加密。

上述的支持并不仅限于通用的 JDBC 实现。如果用于实现 `--`

direct 模式的工具支持 SSL，甚至可以在使用直接连接器的时候加密数据。

下面看看如何使用 SSL 加密 Sqoop 和 MySQL 之间的流量 2。示例 10-17 和示例 10-18 假设 MySQL 已配置 SSL3。如果尚未这样设置，可以从 MySQL 连接器的下载页面（http://dev.mysql.com/downloads/connector/j/5.1.html）下载 MySQL JDBC 驱动。下载完成后，将其安装到一个 Sqoop 可用的位置，如示例 10-17 所示。

> 2Sqoop 示例基于 Kathleen Ting 和 Jarek Jarcec Cecho 合著的 *Apache Sqoop Cookbook*。使用的示例文件和脚本可以从 Apache Sqoop Cookbook 项目页面（http://github.com/jarcec/Apache-Sqoop-Cookbook）获取。

> 3如果 MySQL 尚未配置 SSL，可以按照 MySQL 手册（http://dev.mysql.com/doc/refman/5.7/en/ssl-connections.html）中的说明进行配置。

示例 **10-17**　为 Sqoop 安装 MySQL JDBC 驱动

```
[root@sqoop01 ~]# SQOOP_HOME=/usr/lib/sqoop
[root@sqoop01 ~]# tar -zxf mysql-connector-java-*.tar.gz
[root@sqoop01 ~]# cp mysql-connector-java-*/mysql-connector-java-*-bin.jar
  ${SQOOP_HOME}/lib
[root@sqoop01 ~]#
```

驱动安装到位后，可以使用 Sqoop 的 `list-tables` 命令测试连接，如示例 10-18 所示。

示例 **10-18**　用 list-tables 测试 SSL 连接

```
[alice@sqoop01 ~]$ URI="jdbc:mysql://mysql01.example.com/sqoop"
[alice@sqoop01 ~]$ URI="${URI}?verifyServerCertificate=false"
[alice@sqoop01 ~]$ URI="${URI}&useSSL=true"
[alice@sqoop01 ~]$ URI="${URI}&requireSSL=true"
[alice@sqoop01 ~]$ sqoop list-tables --connect ${URI} \
  --username sqoop -P
Enter password:
cities
countries
normcities
staging_cities
visits
[alice@sqoop01 ~]$
```

设置 MySQL JDBC 驱动使用 SSL 加密的参数，包含在传递给 Sqoop 的 JDBC URI 选项中。

`verifyServerCertificate`

该参数控制客户端是否验证 MySQL 服务器的证书。如果设为 `true`，还需要对参数 `trustCertificateKeyStoreUrl`,`trustCertificateKeyStoreType` 和 `trustCertificateKeyStorePassword` 进行设置。

`useSSL`

该参数设为 `true` 时，客户端会尝试使用 SSL 与 server 进行会话。

requireSSL

该参数设为 `true` 时，若服务器不支持 SSL，客户端会拒绝连接。

示例 10-19 中，试着通过 SSL 导入一个表。

示例 **10-19**　通过 SSL 导入一个 MySQL 表

```
[alice@sqoop01 ~]$ URI="jdbc:mysql://mysql01.example.com/sqoop"
[alice@sqoop01 ~]$ URI="${URI}?verifyServerCertificate=false"
[alice@sqoop01 ~]$ URI="${URI}&useSSL=true"
[alice@sqoop01 ~]$ URI="${URI}&requireSSL=true"
[alice@sqoop01 ~]$ sqoop import --connect ${URI} \
  --username sqoop -P --table cities
Enter password:
...
14/06/27 16:09:07 INFO mapreduce.ImportJobBase: Retrieved 3 records.
[alice@sqoop01 ~]$ hdfs dfs -cat cities/part-m-*
1,USA,Palo Alto
2,Czech Republic,Brno
3,USA,Sunnyvale
[alice@sqoop01 ~]$
```

可以看出，这和在 JDBC URI 中包含 SSL 参数一样简单。可以通过查看作业历史服务器页面中的作业配置确认 SSL 参数在作业执行时被使用，如图 10-2 所示。



图 **10-2**：作业历史服务器页面，展示了 **SSL JDBC** 配置的使用

前例将 `verifyServerCertificate` 设为 `false`。这有助于我们测试，但实际的产品设置中，我们更愿意能够验证正在连接的服务器确实是希望连接的那一台。示例 10-20 将这个参数设为 `true`。

示例 **10-20**　没有 truststore 时证书验证失败

```
[alice@sqoop01 ~]$ URI="jdbc:mysql://mysql01.example.com/sqoop"
[alice@sqoop01 ~]$ URI="${URI}?verifyServerCertificate=true"
[alice@sqoop01 ~]$ URI="${URI}&useSSL=true"
[alice@sqoop01 ~]$ URI="${URI}&requireSSL=true"
[alice@sqoop01 ~]$ sqoop list-tables --connect ${URI} \
  --username sqoop -P
Enter password:
14/06/30 10:52:29 ERROR manager.CatalogQueryManager: Failed to list tables
com.mysql.jdbc.exceptions.jdbc4.CommunicationsException: Communications li

The last packet successfully received from the server was 1,469 millisecon
e last packet sent successfully to the server was 1,464 milliseconds ago.
        at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native M
...
        at org.apache.sqoop.Sqoop.main(Sqoop.java:240)
Caused by: javax.net.ssl.SSLHandshakeException: sun.security.validator.Val
tion: PKIX path building failed: sun.security.provider.certpath.SunCertPat
eption: unable to find valid certification path to requested target
...
[alice@sqoop01 ~]$
```

毫不意外，这样行不通，因为 Java 标准证书的 truststore 不认为我们的 MySQL 服务器证书是可信任证书。诊断这些信任类型问题时，要注意的关键错误消息是 `unable to find valid certification path to requested target`。这通常意味着，服务器没有得到我们受信任证书的签名。最简单的处理方法是，将 MySQL 服务器的证书导入 truststore，并且配置 MySQL JDBC 驱动在连接时使

用该 truststore，如示例 10-21 所示。

示例 **10-21**　通过本地 truststore 列出表

```
[alice@sqoop01 ~]$ keytool \
  -import \
  -alias mysql.example.com \
  -file mysql.example.com.crt \
  -keystore sqoop-jdbc.ts
Enter keystore password:
Re-enter new password:
Owner: EMAILADDRESS=admin@example.com, CN=mysql.example.com, O="Cluster, I
L=San Francisco, ST=California, C=US
Issuer: EMAILADDRESS=admin@example.com, CN=mysql.example.com, O="Cluster,
, L=San Francisco, ST=California, C=US
Serial number: d7f528349bee94f3
Valid from: Fri Jun 27 13:59:05 PDT 2014 until: Sat Jun 27 13:59:05 PDT 20
Certificate fingerprints:
         MD5: 38:9E:F4:D0:4C:14:A8:DF:06:EC:A5:59:76:D1:0C:21
         SHA1: AD:D0:CB:E2:70:C1:89:83:22:32:DE:EF:E5:2B:E5:4F:7E:49:9E:0A
         Signature algorithm name: SHA1withRSA
         Version: 1
Trust this certificate? [no]: yes
Certificate was added to keystore
[alice@sqoop01 ~]$ URI="jdbc:mysql://mysql01.example.com/sqoop"
[alice@sqoop01 ~]$ URI="${URI}?verifyServerCertificate=true"
[alice@sqoop01 ~]$ URI="${URI}&useSSL=true"
[alice@sqoop01 ~]$ URI="${URI}&requireSSL=true"
[alice@sqoop01 ~]$ URI="${URI}&trustCertificateKeyStoreUrl=file:sqoop-jdbc
[alice@sqoop01 ~]$ URI="${URI}&trustCertificateKeyStoreType=JKS"
[alice@sqoop01 ~]$ URI="${URI}&trustCertificateKeyStorePassword=password"
[alice@sqoop01 ~]$ sqoop list-tables --connect ${URI} \
  --username sqoop -P
Enter password:
cities
countries
normcities
staging_cities
visits
[alice@sqoop01 ~]$
```

本例中，首先创建一个包含 MySQL 服务器证书的 truststore，然后将 MySQL JDBC 驱动指向该 truststore。这要求在 JDBC URI 中设置一些额外的参数，如下所示。

`trustCertificateKeyStoreUrl`

　　一个指向 keystore 位置的 URL，该 keystore 用于验证 MySQL 服务器证书。

`trustCertificateKeyStoreType`

　　用于验证 MySQL 服务器证书的 keystore 类型。

`trustCertificatekeyStorePassword`

　　用于验证 MySQL 服务器证书的 Keystore 密码。

注意，我们使用了一个相对路径 file:<URI> 指定 truststore 的位置，它在下一个示例中很重要。列出表之后，试着在示例 10-22 中进行一次导入。

示例 **10-22**　通过 truststore 导入表

```
[alice@sqoop01 ~]$ URI="jdbc:mysql://mysql01.example.com/sqoop"
[alice@sqoop01 ~]$ URI="${URI}?verifyServerCertificate=true"
[alice@sqoop01 ~]$ URI="${URI}&useSSL=true"
[alice@sqoop01 ~]$ URI="${URI}&requireSSL=true"
[alice@sqoop01 ~]$ URI="${URI}&trustCertificateKeyStoreUrl=file:sqoop-jdbc
[alice@sqoop01 ~]$ URI="${URI}&trustCertificateKeyStoreType=JKS"
[alice@sqoop01 ~]$ URI="${URI}&trustCertificateKeyStorePassword=password"
[alice@sqoop01 ~]$ sqoop import \
  -files sqoop-jdbc.ts \
  --connect ${URI} \
  --username sqoop \
  -P \
  --table cities
Enter password:
...
14/06/30 10:57:13 INFO mapreduce.ImportJobBase: Retrieved 3 records.
[alice@sqoop01 ~]$
```

与上一个列出表的示例相比，变化不是很大。除了使用了一样的 URI 之外，我们还增加了一个 -file 命令行参数。-file 选项会将文件列表放入 Hadoop 的分布式缓存，该分布式缓存会把文件复制到集群中的每一个节点，并将其放置到正运行任务的工作目录下。这项功能很有用，因为这意味着我们给 trustCertificateKeyStoreUrl 的配置对本地机器和执行任务的所有节点都同样有效，这也正是为什么我们想用 truststore 作为启动 Sqoop 作业的工作目录。

加密支持并不仅限于普通模式，像 MySQL 的直接模式就使用了支持 SSL 的 mysqldump 和 mysqlimport 工具。示例 10-23 在直接模式中启用 SSL。

示例 10-23　使用直接模式通过 truststore 导入表

```
[alice@sqoop01 ~]$ URI="jdbc:mysql://mysql01.example.com/sqoop"
[alice@sqoop01 ~]$ URI="${URI}?verifyServerCertificate=true"
[alice@sqoop01 ~]$ URI="${URI}&useSSL=true"
[alice@sqoop01 ~]$ URI="${URI}&requireSSL=true"
[alice@sqoop01 ~]$ URI="${URI}&trustCertificateKeyStoreUrl=file:sqoop-jdbc
[alice@sqoop01 ~]$ URI="${URI}&trustCertificateKeyStoreType=JKS"
[alice@sqoop01 ~]$ URI="${URI}&trustCertificateKeyStorePassword=password"
[alice@sqoop01 ~]$ sqoop import \
  -files sqoop-jdbc.ts,mysql.example.com.crt \
  --connect ${URI} \
  --username sqoop \
  -P \
  --table cities \
  --direct \
  -- \
    --ssl \
    --ssl-ca=mysql.example.com \
    --ssl-verify-server-cert
Enter password:
...
14/06/30 15:32:43 INFO mapreduce.ImportJobBase: Retrieved 3 records.
[alice@sqoop01 ~]$
```

同样，这与上一个示例也很相似。主要的区别在于，我们将 `mysql.example.com.crt` 添加到了 `-file` 选项，因而节点会拥有 `mysqldump` 工具要求的 PEM 格式的证书。还添加了 `--direct` 选项以启用直接模式。最后，增加了 `--ssl`、`--ssl-ca=mysql.example.com` 和 `--ssl-verify-server-cert` 选项。`--` 标记意味着其后所有参数应当被传给实现直接模式的工具。剩余的参数将会被 `mysqldump` 用于启用 SSL、设置 CA 证书位置以及让 `mysqldump` 验证 MySQL 服务器的证书。

10.3　导入工作流

目前，我们已经了解了数据通常如何导入 Hadoop 环境，也了解了导入管道的机密性、完整性和可用性的不同选项。然而，导入数据是一个典型的全局 ETL（抽取、转换和加载）过程，因此在全局 ETL 流的上下文环境中必须进行额外的考虑。

我们之前忽略的一个细节是，Sqoop 这类工具是从哪里启动的。通常情况下，你会希望限制用户能够访问的接口。正如在 3.4.1 节描述的那样，有很多种访问远程协议的途径能够得到保护，并且具体的架构

依赖于用户的需求。最常见的限制边界服务（包括导入）的方法是，在边界节点上部署 Flume 和 Sqoop 这类服务。边界节点只不过是既能访问内部 Hadoop 集群网络、又能访问外部网络的服务器。典型的集群部署将会通过主机或网络防火墙限制对特殊主机的特殊端口的访问。数据导入场景中，我们能在执行并行导入时利用仍然部署着数据拉取机制（例如 Sqoop）的边界节点，限制向 Hadoop 集群推送数据，从而避免向外部暴露敏感的 Hadoop 服务。

“只允许远程登录边界节点”在很大程度上降低了用户物理登入 Hadoop 集群的风险，这使得能在减少潜在危险角色数量的同时，能够将精力集中在监测和安全审计工作上。生成数据流时，通常应设置专用 ETL 账户或组执行全局的工作流。对于那些需要精确审计的组织，推荐使用相互独立的用户账户发起操作，这样能使操作行为更容易追踪到个人。

除了 Flume 和 Sqoop 之外，边界节点还可能运行有代理服务或其他远程用户协议。例如，HDFS 支持 HttpFS 代理服务，该服务会向外开放一个供 HDFS 使用的读 / 写 REST 接口。

与 HDFS 一样，HttpFS 完全支持基于 Kerberos 的身份验证和 HDFS 内建的权限控制。在一个边界节点上运行 HttpFS，能够允许对 HDFS 中存储的数据进行有限的访问，甚至可以被用于某些数据的导入。

另一个常见的边界节点服务是 Oozie。Oozie 是一个规划和执行工作流的工具。由 Sqoop 作业、Hive 查询、Pig 脚本以及 MapReduce 作业组成的复杂工作流可以被组合成独立的单元，并且通过 Oozie 被可靠地规划和执行。Oozie 还提供了一个支持基于 Kerberos 身份验证的 REST 接口，并能安全地向边界节点开放。

一些用例中，有必要在把文件推送到 HDFS、HBase 或 Accumulo 之前，将其放置在一个边界节点上。创建这些本地磁盘（有时是 NFS 挂载的磁盘）目录时，应使用标准的操作系统控制手段，将对数据的访问限制为仅限授权用户。这里要再次强调，应当定义一个或多个 ETL 组，并且将对原始数据的访问范围限制为相对受信任的组。

10.4　企业架构

对于数据导入的这些讨论帮助我们明确了这样一个有用的观点：Hadoop 从来不是部署在真空之中的。Hadoop 不可避免地要与已存在的和正在演变的企业架构相结合，这意味着，不能只考虑 Hadoop

自身安全。决定如何保护集群时，必须研究已经应用于数据和系统的需求，这些需求由企业安全标准、威胁模型和特殊数据的集敏感度所驱动。需要特别指出的是，如果数据源本身就毫无安全可言，那么封锁 Hadoop 集群或集群的数据导入管道是毫无意义的。

这与企业引进数据仓库系统的情况一样。一个典型的部署中，应用程序和支持它们的事务处理系统是紧耦合的。这使得安全集成简单而直接，因为对后端数据库的访问通常被限制在生成和处理这些数据的应用程序中。一旦事务性数据重要到需要备份时，一些安全细节就被重视起来。然而，这仍然能算得上相对简单的集成，因为可以限制备份数据仅能被那些已拥有对源系统访问权的受信任管理员访问。

试图把数据从事务处理系统移动至分析型数据仓库，以便独立执行分析应用时，事情变得有趣起来。若使用传统的数据仓库系统，需要将事务数据库的安全配置与新数据仓库的特性进行对比。一旦数据被导入仓库，该机制就会工作得很好，并且可以将同样的分析工作应用于基于数据库授权的 Sentry。然而，想知道如何处理事务系统和分析平台之间的数据时，必须加以注意。

对这些传统系统的使用可以归结为保护 ETL 网格，正是这些 ETL 网格将数据加载到数据仓库。很明显，对 ETL 网格所做的考虑同样适用于 Hadoop 集群的导入管道，特别是必须考虑：在何时何地的加密措施对保护数据机密性来说才是必要的。还需要对维持数据的完整性格外加以重视。在传统的 ETL 网络中，这尤其重要，因为这种传统网格可能没有足够的存储容量在传输之后仍保留原始数据。最后，还需要对 ETL 网格的可用性加以注意，从而确保不会影响源系统或数据仓库满足用户要求的性能。这正与之前讨论的过程完全一样，即导入数据到 Hadoop 的一般过程和使用 Flume 和 Sqoop 的特定过程。

此外，需要重申，这个工作是双向的，仅在 Hadoop 导入或进行不属于源系统的查询时才应用安全控制策略是不合理的。正如在已经开始进行已有事务性或分析性工具的安全保障设计工作后，却留着 Hadoop 门户大开一样不合理。考虑所有因素的最佳时机是在设计导入管道的时候，因为这正是 Hadoop 与其余企业架构融为一体的地方，是比对安全和威胁模型，并仔细考虑整体 Hadoop 部署的安全架构的完美时机。

10.5　小结

本章集中于数据从外部源到 Hadoop 的传输。简单讨论批量文件导入

后，我们继续聚焦于使用 Flume 进行基于事件的数据导入，以及使用 Sqoop 从关系型数据库导入数据。可以发现，这些通用的数据导入机制能保护传输中的数据完整性。本章的一个关键点是，集群内部对数据的保护需要延伸至从数据源导入的全部过程。这种保护模式应当与源系统的保护等级相匹配。

我们已经了解了数据导入和集群内部数据的保护，下面学习数据保护的最后一项内容：保护数据提取过程和客户端的访问。

第 11 章　数据提取和客户端访问安全

Hadoop 的核心宗旨之一是，将处理工作带到数据所在之处，而不是反其道行之。因此，我们的焦点一直都是“集群内的安全如何实现”。本章将讲解保护 Hadoop 集群的最后一部分内容，也就是保护客户端的访问和数据提取的过程。虽然对 Hadoop 数据的大多数处理过程在集群内完成，但用户是通过外部工具访问数据的，并且在一些诸如企业数据仓库的用例中，还可能需要专用工具以提取海量数据。

客户端访问的最基本形式是命令行工具。正如 3.3.4 节的“边界节点”，集群中，将外部访问限制在很小的一个边界节点集合中是很常见的。用户使用 `ssh` 远程登录一个边界节点，并且使用多种命令行工具与集群交互。表 11-1 是最常用的一些命令的简要描述。

表**11-1**：客户端访问的常用命令行工具

命令	描述
`hdfs dfs -put <src> <dst>`	将本地文件复制到 HDFS
`hdfs dfs -get <src> <dst>`	从 HDFS 下载文件到本地系统
`hdfs dfs -cat <path>`	打印文件内容到标准输出
`hdfs dfs -ls <path>`	列出某路径下的文件和目录
`hdfs dfs -mkdir <path>`	在 HDFS 中创建一个目录
`hdfs dfs -cp <src> <dst>`	把 HDFS 中的文件复制到一个新的地址
`hdfs dfs -mv <src> <dst>`	把 HDFS 中的文件移动到一个新的地址
`hdfs dfs -rm <path>`	从 HDFS 中删除一个文件
`hdfs dfs -rmdir <path>`	从 HDFS 中删除一个目录
`hdfs dfs -chgrp <group> <path>`	改变一个文件或目录的组
`hdfs dfs -chmod <mode> <path>`	改变一个文件或目录的权限
`hdfs dfs -chown <owner>[:<group>] <path>`	改变一个文件或目录的拥有者
`yarn jar <jar> [<main-class>] <args>`	运行一个 JAR 文件，通常用于启动 MapReduce 作业或其他 YARN 作业
`yarn application -list`	列出正在运行的 YARN 应用
`yarn application -kill <app-id>`	终止一个 YARN 应用
`mapred job -list`	列出一个正在运行的 MapReduce 作业
`mapred job -status <job-id>`	获取一个 MapReduce 作业的状态
`mapred job -kill <job-id>`	终止一个 MapReduce 作业
`hive`	启动一个 Hive SQL shell（不推荐；推荐使用 beeline）
`beeline`	给 Hive 或 Impala 启动一个 SQL shell
`impala-shell`	启动一个 Impala SQL shell
`hbase shell`	启动一个 HBase shell
`accumulo shell`	启动一个 Accumulo shell
`oozie`	运行、检查和终止 Oozie 作业
`sqoop export`	从 HDFS 导出表到数据库
`sqoop import`	从数据库导入表到 HDFS

11.1　Hadoop命令行接口

核心的 Hadoop 命令行工具（hdfs、yarn 和 mapred）只支持 Kerberos 或者使用委托令牌的方法进行身份验证。验证这些命令最简单的方式是，在命令执行前使用 kinit 取得 Kerberos 的票据授予票据（TGT）。1 如果执行 Hadoop 命令之前没有获得 TGT，你将会看到示例 11-1 所示的相似错误，特别是 failed to find any Kerberos tgt。

1回顾表 4-1 的 TGT 相关内容。

示例 **11-1**　在没有 Kerberos TGT 的情况下执行 Hadoop 命令

```
[alice@hadoop01 ~]$ hdfs dfs -cat movies.psv
cat: Failed on local exception: java.io.IOException: javax.security.sasl.
SaslException: GSS initiate failed [Caused by GSSException: No valid crede
provided (Mechanism level: Failed to find any Kerberos tgt)]; Host Details
host is: "hadoop01.example.com/172.25.2.196"; destination host is: "hadoop
example.com":8020;
```

下面看看 Alice 先使用 kinit 取得 TGT 之后的情况（示例 11-2）。

示例 **11-2**　在 kinit 之后执行 Hadoop 命令

```
[alice@hadoop01 ~]$ kinit Password for alice@EXAMPLE.COM: [alice@hadoop01
hdfs dfs -cat movies.psv 1|Toy Story (1995)|01-Jan-1995||http://us.imdb.co
titleexact?Toy%20Story%20( ...
```

这次命令被成功执行，并且打印文件 movies.psv 中的内容。使用 Kerberos 进行身份验证的好处是，用户不需要为每个命令单独进行验证。如果在会话开始时使用 kinit，或将 Linux 系统配置为登录时获取 Kerberos TGT，则可以运行任意数量的 Hadoop 命令，所有身份验证工作将在幕后完成。

虽然通常不这么做，但命令行工具可以使用委托令牌进行 HDFS 的身份验证。为了获得委托令牌，需要通过 Kerberos 的身份验证。HDFS 提供了一种命令行工具，以获取委托令牌并保存到一个文件。设置环境变量 HADOOP_TOKEN_fiLE_LOCATION 之后，该令牌就能被用

于随后的 HDFS 命令。委托令牌的使用如示例 11-3 所示。

示例 **11-3**　使用委托令牌执行 Hadoop 命令

```
[alice@hadoop01 ~]$ kinit
Password for alice@EXAMPLE.COM:
[alice@hadoop01 ~]$ hdfs fetchdt --renewer alice nn.dt
14/10/21 19:19:32 INFO hdfs.DFSClient: Created HDFS_DELEGATION_TOKEN token
 alice on 172.25.3.210:8020
Fetched token for 172.25.3.210:8020 into file:/home/alice/nn.dt
[alice@hadoop01 ~]$ kdestroy
[alice@hadoop01 ~]$ export HADOOP_TOKEN_FILE_LOCATION=nn.dt
[alice@hadoop01 ~]$ hdfs dfs -cat movies.psv
1|Toy Story (1995)|01-Jan-1995||http://us.imdb.com/M/title-exact?Toy%20Sto
...
```

委托令牌同基于 Kerberos 的身份验证是完全独立的。委托令牌一旦被分发，默认具有 24 小时的有效期，并且可以被续至最多 7 天。可以通过设置 `dfs.namenode.delegation.token.renewal-interval` 改变令牌的初始有效期，该参数表示了令牌需要续租前还剩下的毫秒数。可以设置 `dfs.namenode.delegation.token.max-lifetime` 改变令牌的最大更新周期，该配置参数的单位也是毫秒。这些令牌独立于 Kerberos 系统，所以如果 Kerberos 凭证在 KDC 中是激活的，委托令牌会在它们的指定生命周期中始终有效，没有任何管理方法能强制取消委托令牌。

Hadoop 命令行工具没有单独的授权模型，而依赖于集群的配置控制哪些内容是用户能够访问的。要复习 Hadoop 身份验证，可回顾 6.1 节~6.3 节。

11.2　保护应用安全

开发一个应用时，在设计方面会遇到许多选择，例如为用户接口使用什么框架，或者为后端存储使用什么系统等。应用设计中最关键的要素是，如何保护一个应用的安全性。这个问题通常由于与应用交互的大量接口而变得更加复杂。

涉及数据的授权应当在哪里进行时，通常有两种流派。一种观点认为，安全问题（特别是授权）应属于应用层。这是有道理的，因为应用程序通常与数据的控制策略有更多上下文关联。但此观点的缺点在于，这意味着每一个应用程序都必须重新实现通用的服务。随着时间

的推移，数据库获得了发展，因此，应用程序将授权标签下放至数据库时，数据库可以存储和强制执行授权控制。这是给 Accumulo 添加单元级的可见性标签，和给 HBase 添加单元级的访问控制表（ACL）和可见性标签的主要动机之一。

与“授权决策在哪里进行”紧密相关的是，用户账户能深入到哪个地步？历史上，数据库一直保持着它们的私有身份目录，这通常使尝试和复制数据库目录中的用户变得复杂。

Accumulo 仍然使用自己的身份目录，因此它也具有这一缺陷。为了应对这个问题，许多应用程序的开发者采用了这样一种模式：使用数据库存储应用级账户，并由应用程序负责降级访问，以此利用数据库级的授权。

降级访问的功能要求，用户使用 Java API 访问 Accumulo 时，需要通过一系列的授权。这使得应用程序能对终端用户执行身份验证，并使用集中服务查找用户的授权。随后，应用程序会在读取数据时传递这些终端用户的授权，Accumulo 会自动将应用程序的授权信息与终端用户的授权信息交叉比对。这意味着，给终端用户提供基于他们个体级别的更细粒度的访问控制时，也可以对应用程序可访问数据的最高级别进行控制。

最后，终端用户的授权能有多深入取决于应用程序开发者的决策。随着时间的推移，应用程序和基于 Hadoop 的数据存储被集成到同一个身份目录下之后，应用程序开发者想要做出明智的设计会更加容易。安全领域并不是一成不变的，而是在持续地发展。

11.3 HBase

第 5 章中，我们了解过如何配置 HBase 以使用 Kerberos 作为身份验证机制。HBase 客户端可以通过控制台、Java API 或某个 HBase 的网关服务访问 HBase。所有的客户端访问 API 都支持 Kerberos 作为身份验证机制，且要求用户在连接前先取得 Kerberos 的 TGT。

这种客户端的访问方式依赖于具体的使用场景。对于数据库的管理访问行为，例如创建、修改或删除表，通常使用 HBase 控制台完成；使用 MapReduce 或其他数据处理框架时，通常使用 Java API；其他类型的 HBase 应用也可以直接使用 Java API，或通过网关服务访问 HBase。使用非 Java 语言访问 HBase 时，选择网关 API 的情况非常普遍。网关方式还为管理员提供了一个遏止点，以限制对 HBase 的直

接访问。接下来，讨论如何安全配置 HBase 网关之前，先看看怎样通过控制台与 HBase 进行安全交互。

11.3.1 HBase shell

使用控制台时，用户通常在执行 kinit 命令之前就获得了 TGT。如果试图在没有运行 kinit 之前就运行控制台，将会如示例 11-4 所示。需要注意的是，栈跟踪信息末尾的错误信息 `Failed to find any Kerberos tgt`。

示例 11-4　没有 Kerberos TGT 时使用 HBase 控制台

```
[alice@hadoop01 ~]$ hbase shell
14/11/13 14:45:53 INFO Configuration.deprecation: hadoop.native.lib is dep
Instead, use io.native.lib.available HBase Shell; enter 'help<RETURN>' for
supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.6, rUnknown, Sat Oct 11 15:15:15 PDT 2014

hbase(main):001:0> list
TABLE
14/11/13 14:46:00 WARN ipc.RpcClient: Exception encountered while connecti
 to the server : javax.security.sasl.SaslException: GSS initiate failed [C
sed by GSSException: No valid credentials provided (Mechanism level: Faile
to find
any Kerberos tgt)]
14/11/13 14:46:00 FATAL ipc.RpcClient: SASL authentication failed. The mos
cause is missing or invalid credentials. Consider 'kinit'.
javax.security.sasl.SaslException: GSS initiate failed [Caused by GSSExcep
on: No valid credentials provided (Mechanism level: Failed to find any Ker
...

ERROR: No valid credentials provided (Mechanism level: Failed to find any
Kerberos tgt)

Here is some help for this command:
List all tables in hbase. Optional regular expression parameter could
be used to filter the output. Examples:

  hbase> list
  hbase> list 'abc.*'
  hbase> list 'ns:abc.*'
  hbase> list 'ns:.*'


hbase(main):002:0>
```

现在在示例 11-5 中再试一遍，但这次会在执行控制台前先使用 kinit 获取 TGT。

示例 **11-5** 执行 kinit 后使用 HBase 控制台

```
[alice@hadoop01 ~]$ kinit
Password for alice@EXAMPLE.COM:
[alice@hadoop01 ~]$ hbase shell
14/11/13 14:53:56 INFO Configuration.deprecation: hadoop.native.lib
 is deprecated. Instead, use io.native.lib.available
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.6, rUnknown, Sat Oct 11 15:15:15 PDT 2014

hbase(main):001:0> list
TABLE
analytics_demo
document_demo
2 row(s) in 3.1900 seconds

=> ["analytics_demo", "document_demo"]
hbase(main):002:0> whoami
alice@EXAMPLE.COM (auth:KERBEROS)
    groups: alice, hadoop-users

hbase(main):003:0>
```

HBase 控制台没有独立的授权配置，所有访问都必须通过 HBase 的授权配置进行授权，详见 6.6 节。

11.3.2 HBase REST网关

HBase 具有两种网关服务实现方式：REST 服务和 Thrift 服务。这两种实现方法都允许使用非 Java 语言访问 HBase，并且都支持身份验证、身份模拟和通过加密实现的机密性。部署哪种网关应当依赖于应用程序开发者的具体需求：通过基于 JavaScript 的 Web 应用程序直接访问网关时，通常使用 REST 接口；通过其他语言的访问则通常使用 Thrift API。现在看看怎样配置 REST 网关的身份验证功能。

第一步是为 REST 网关创建一个 Kerberos 规则，以和 HBase 的其余部分对话。这是一个服务主体，应该包括运行 REST 网关的服务器主机名——例如 `rest/rest.example.com@EXAMPLE.COM`，其中 `rest.example.com` 由运行有 REST 网关的服务器的完全限定域名代替。创建服务主体并导出含有主体密钥的文件后，需要配置 REST 服务，使其通过 Kerberos 与 HBase 集群交互。先看下面的 `hbase-`

site.xml 文件：

```
<property>
  <name>hbase.rest.keytab.file</name>
  <value>/etc/hbase/conf/hbase-rest.keytab</value>
</property>
<property>
  <name>hbase.rest.kerberos.principal</name>
  <value>rest/_HOST@EXAMPLE.COM</value>
</property>
```

如果启用 HBase 授权机制，则还需要为 REST 服务使用的主体创建一个顶层 ACL。假设想授予通过 REST 网关访问的一切权限（包括管理访问权限），那么应当使用 HBase 控制台执行下列内容（详见 6.6 节）：

```
hbase(main):001:0> list grant 'rest', 'RWCA'
```

如果使用了不同的主体名，那么应当用主体的短名称代替 rest。下一步是通过 SPNEGO/Kerberos 启用 REST 客户端的验证机制。对于每一个 SPNEGO 的协议规范，需要创建一个符合 HTTP/rest.example.com@EXAMPLE.COM 格式的主体，其中 rest.example.com 用运行有 REST 网关的服务器的完全限定域名代替。对 hbase-site.xml 进行如下设置，以启用验证机制：

```
<property>
  <name>hbase.rest.authentication.type</name>
  <value>kerberos</value>
</property>
<property>
  <name>hbase.rest.authentication.kerberos.principal</name>
  <value>HTTP/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>hbase.rest.authentication.kerberos.keytab</name>
  <value>/etc/hbase/conf/hbase-rest.keytab</value>
</property>
```

本例中，将 REST 的验证密钥表与 HBase 的验证密钥表设置为同一位置。这意味着或者需要将两种 keytab 同时导出，或者使用 ktutil 将两种主体的 keytab 合并为一个 keytab 文件。另外，还可以为 REST 客户端验证和 HBase 验证机制使用不同的 keytab 文件。

REST 服务通常使用 hbase.rest.kerberos.principal 同 HBase 进行身份验证，但它是代表同 REST 服务进行身份验证的用户执行操作的。为了做到这些，REST 服务必须拥有模拟其他用户的权限。我们可以使用与 5.2.4 节相同的 hadoop.proxyuser.<proxy>.groups 和 hadoop.proxyuser.<proxy>.hosts 的设置。这些设置能控制哪些用户可由代理模拟，以及它们可以从哪些主机进行身份模拟。这些设置的值是由逗号分隔的组和主机列表，* 号代表所有组 / 主机。例如，如果希望 rest 用户能模拟来自 hbase-user 组中任意主机的任意用户，需要将下列内容添加到 HBase Master 中的 hbase-site.xml 文件：

```
<property>
  <name>hadoop.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hadoop.proxyuser.rest.groups</name>
  <value>hbase-users</value>
</property>
<property>
  <name>hadoop.proxyuser.rest.hosts</name>
  <value>*</value>
</property>
```

REST 服务还支持远程 REST 客户端模拟终端用户，这称为双层用户模拟，因为被 REST 客户端模拟的用户是由 REST 服务模拟的。这允许运行一个通过 REST 服务访问 HBase 的应用程序，该应用程序可以通过所有途径传递用户凭证。要想启用这种身份模拟，可将 hbase.rest.support.proxyuser 的值设为 true。可以对 hadoop.proxyuser.<app user>.groups 进行配置，控制访问 REST 服务的应用程序能模拟哪些终端用户。比如有个 whizbang 应用程序，可以模拟 whizbang-users 组内的任意用户，那么应该对 REST 服务中的 hbase-site.xml 文件进行如下设置：

```
<property>
  <name>hbase.rest.support.proxyuser</name>
  <value>true</value>
</property>
<property>
  <name>hadoop.security.authorization</name>
  <value>true</value>
</property>
<property>
```

```
  <name>hadoop.proxyuser.whizbang.groups</name>
  <value>whizbang-users</value>
</property>
<property>
  <name>hadoop.proxyuser.whizbang.hosts</name>
  <value>*</value>
</property>
```

图 11-1 展示了双层模拟是如何通过 HBase REST 服务工作的。终端用户 Alice 通过 LDAP 用户名和口令进行身份验证，对 Hue 证明了她的身份 (1)。然后 Hue 通过 Kerberos 使用 `hue/hue.example.com@EXAMPLE.COM` 主体进行身份验证，并传递了 `alice` 的 `doAs` 用户身份 (2)。最后，HBase REST 服务通过 Kerberos 使用 `rest/rest.example.com@EXAMPLE.COM` 主体进行身份验证，并传递了 `alice` 的 `doAs` 用户身份 (3)。该流程在从用户到 HBase 的整条路径中有效传递了 Alice 的身份凭据。

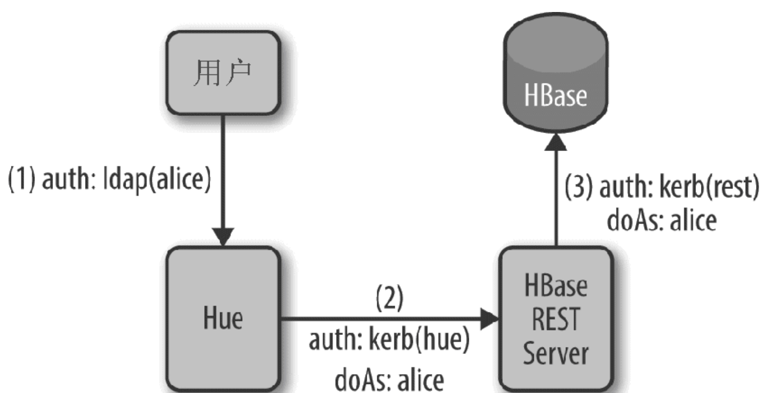


图 **11-1**：双层用户模拟

REST 服务可通过启用 TLS/SSL 支持客户端和 REST server 间的加密连接。要想启用 SSL，可以将 `hbase.rest.ssl.enabled` 的值设为 `true`，并将 REST 服务配置为通过私钥和证书使用 Java keystore 文件。如果 keystore 文件在 `/etc/hbase/conf/rest.example.com.jks`，并且密钥和 keystore 使用 `secret` 作为口令，那么应按照如下内容对 REST 服务的 `hbasesite.xml` 文件进行配置：

```
<property>
  <name>hbase.rest.ssl.enabled</name>
  <value>true</value>
```

```
</property>
<property>
  <name>hbase.rest.ssl.keystore.store</name>
  <value>/etc/hbase/conf/rest.example.com.jks</value>
</property>
<property>
  <name>hbase.rest.ssl.keystore.password</name>
  <value>secret</value>
</property>
<property>
  <name>hbase.rest.ssl.keystore.keypassword</name>
  <value>secret</value>
</property>
```

如果 REST 服务的证书没有被某个已被信任的证书签名，那么需要使用命令行工具 `keytool` 将该证书导入 Java 的 central truststore：

```
[hbase@rest ~]$ keytool -import -trustcacerts -file rest.example.com.crt \
  -keystore $JAVA_HOME/jre/lib/security/cacerts
```

 上述 keytool 命令能将证书导入 Java 的 central truststore。这意味着导入的任何证书都会被任意 Java 应用程序所信任——不仅仅是 HBase，而是使用了该 JRE 的所有应用。

11.3.3 HBase Thrift网关

与 REST 网关类似，HBase Thrift 网关支持用户通过 Kerberos 进行身份验证。第一步是为 Thrift 网关创建一个 Kerberos 主体，与 HBase 其余部分交互。这是一个服务主体，并且应该包含运行 Thrift 网关的服务器主机名（如 `thrift/thrift.example.com@EXAMPLE.COM`）。创建服务主体并导出含有主体密钥的 keytab 文件后，需要配置 Thrift 服务，使其通过 Kerberos 与 HBase 集群交互。看看下面的 `hbase-site.xml` 文件：

```
<property>
  <name>hbase.thrift.keytab.file</name>
  <value>/etc/hbase/conf/hbase.keytab</value>
</property>
<property>
  <name>hbase.thrift.kerberos.principal</name>
  <value>thrift/_HOST@EXAMPLE.COM</value>
</property>
```

如果启用 HBase 的授权机制，则还需要为 Thrift 服务使用的主体创建一个顶层 ACL（访问控制表）。假设想授予通过 Thrift 网关的一切访问权限（包括管理访问权限），那么应当使用 HBase shell 执行下列内容：

```
hbase(main):001:0> list grant 'thrift', 'RWCA'
```

此时，Thrift 网关能够访问 HBase 集群，但不能进行用户身份验证。还需要将 `hbase.thrift.security.qop` 的值设为以下 3 个值之一，以启用身份验证。

`auth`

启用身份验证。

`auth-int`

启用身份验证和完整性检查。

`auth-conf`

启用身份验证、机密性机制（即加密）和完整性检查。

与 REST 网关类似，需要启用 Thrift 用户以模拟那些同 Thrift 网关进行身份验证的用户。同样地，将会用到 HBase Master 中 `hbase-site.xml` 文件的 `hadoop.proxyuser` 配置：

```
<property>
  <name>hadoop.security.authorization</name>
  <value>true</value>
</property>
<property>
  <name>hadoop.proxyuser.thrift.groups</name>
  <value>hbase-users</value>
</property>
<property>
  <name>hadoop.proxyuser.thrift.hosts</name>
  <value>*</value>
</property>
```

与 REST 网关不同的是，Thrift 网关不支持应用程序用户模拟终端用

户。这意味着，如果一个应用程序通过 Thrift 网关访问 HBase，那么其所有访问行为都会被作为 app 用户得到处理。即将到来的 HBase 1.0 添加了一项功能：Thrift 网关的传输方式被配置为 HTTPS 时的身份模拟。这方面的工作可以在 HBASE-12640（http://issues.apache.org/jira/browse/ HBASE-12640）中找到。

图 11-2 展示了 Thrift 网关是如何进行身份模拟的。假设有一个通过 Thrift 网关访问 HBase 的 Web 应用。用户 Alice 通过 Web 应用使用 PKI（公钥基础设施）进行身份验证 (1)，然后应用程序使用 Kerberos 与 Thrift 网关进行身份验证 (2)。由于仅支持单层身份模拟， Thrift 网关使用 `thrift/thrift.example.com@EXAMPLE.COM` 主体和 `app` 的 `doAs` 与 HBase 进行身份验证 (3)。HBase 不会知道原始的终端用户是 Alice，并且向 Alice 展示结果之前，由这个 Web 应用负责申请其他授权控制。请回顾图 11-1，并比对单层和双层的用户模拟。

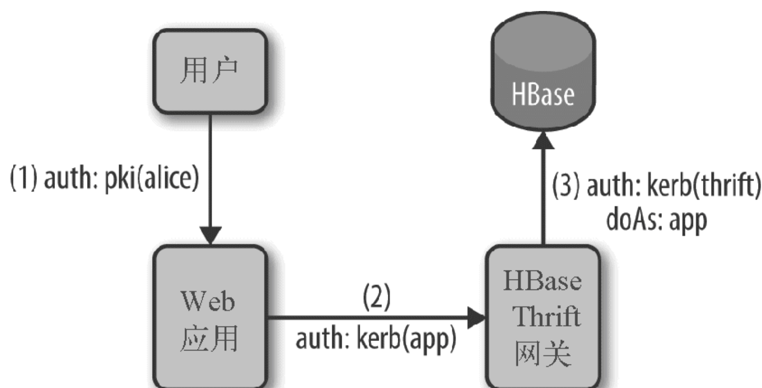


图 **11-2：Thrift** 网关的应用层身份模拟

11.4 Accumulo

Accumulo 客户端的访问可以通过两种机制完成：控制台和代理服务。Accumulo 控制台与 HBase 控制台类似，然而 Accumulo 的代理服务却和 HBase 的 thrift 网关服务类似。

11.4.1 Accumulo shell

与 HBase 不同的是，Accumulo 使用用户名和口令进行身份验证。支持客户端 Kerberos 验证的机制会在 Accumulo 1.7.0 版中到来，可以在 ACCUMULO-1815（http://issues.apache.org/jira/browse/

ACCUMULO-2815）中对其进行跟踪。这意味着，客户端连接 Accumulo 时，必须同时提供用户名和口令。使用 Accumulo 控制台时，可以使用 -u 或 --user 命令行参数传递用户名，或者可以默认使用正在运行 shell 的 Linux 用户名。如果不传递任何参数，Accumulo 会主动要求在标准输入中输入口令。另外，还可以使用命令行、文件或者环境变量提供口令，这些方法可分别通过参数 -p 或 --password 的 pass:<lit eral password>、file:<path to file with password> 或者 env:<environment variable with password> 选项得到启用。还可以在运行 Accumulo 控制台之后，使用 user 命令改变当前用户，新用户将会被要求输入口令。在 Accumulo 控制台中传递口令的多种方法见示例 11-6。注意，提供错误口令时，控制台会打印 Username or Password is Invalid 消息。

示例 11-6　通过 Accumulo 控制台进行身份验证

```
[alice@hadoop01 ~]$ accumulo shell
Password: ***
2014-11-13 15:19:54,225 [shell.Shell] ERROR: org.apache.accumulo.core.clie
.AccumuloSecurityException: Error BAD_CREDENTIALS for user alice - Usernam
or Password is Invalid
[alice@hadoop01 ~]$ accumulo shell
Password: ******

Shell - Apache Accumulo Interactive Shell
-
- version: 1.6.0
- instance name: accumulo
- instance id: 382edcfb-5078-48b4-8570-f61d92915015
-
- type 'help' for a list of available commands
-
alice@accumulo> quit
[alice@hadoop01 ~]$ accumulo shell -p pass:secret

Shell - Apache Accumulo Interactive Shell
-
- version: 1.6.0
- instance name: accumulo
- instance id: 382edcfb-5078-48b4-8570-f61d92915015
-
- type 'help' for a list of available commands
-
alice@accumulo> quit
[alice@hadoop01 ~]$ accumulo shell -p file:accumulo_pass.txt

Shell - Apache Accumulo Interactive Shell
-
```

```
- version: 1.6.0
- instance name: accumulo
- instance id: 382edcfb-5078-48b4-8570-f61d92915015
-
- type 'help' for a list of available commands
-
alice@accumulo> quit
[alice@hadoop01 ~]$ accumulo shell -p env:ACCUMULO_PASS

Shell - Apache Accumulo Interactive Shell
-
- version: 1.6.0
- instance name: accumulo
- instance id: 382edcfb-5078-48b4-8570-f61d92915015
-
- type 'help' for a list of available commands
-
alice@accumulo> user bob
Enter password for user bob: ***
bob@accumulo>
```

Accumulo 控制台没有属于自己的授权配置机制，所有访问会由 Accumulo 的授权配置机制进行授权，详见 6.6 节。

11.4.2　Accumulo代理服务

Accumulo 有一个代理服务与 HBase 的 Thrift 网关很相似，该代理服务可以被部署在任意能使用 Java 客户端 API 的服务器上。具体来说，这意味着该服务器必须能够与 Accumulo Master、ZooKeeper quorum、NameNode 和 DataNode 进行通信。对 Accumulo 代理服务的配置在 `$ACCUMULO_HOME/proxy/proxy.properties` 文件中进行，参数 `protocolFactory` 决定了服务器和客户端使用的下层 Thrift 协议。如果要改变该参数的设置，必须与客户端正在使用的协议实现相符合。如果需要支持多种 Thrift 协议，则应当部署多个代理服务。

其他必须在客户端和代理服务之间同步的设置是 `tokenClass`。代理服务不会直接对客户端进行身份验证，而会将用户提供的身份验证令牌传给 Accumulo。若需要同时支持多种身份验证令牌，也需要部署多个代理服务。

`proxy.properties` 文件的示例如下：

```
protocolFactory=org.apache.thrift.protocol.TCompactProtocol$Factory
tokenClass=org.apache.accumulo.core.client.security.tokens.PasswordToken
```

```
port=42424
instance=accumulo-instance
zookeepers=zoo-1.example.com,zoo-2.example.com,zoo-3.example.com
```

由于 Accumulo 会把身份验证令牌从访问代理服务的应用传递给 Accumulo，这相当于一个单层身份模拟，如图 11-2 所示。Accumulo 支持应用程序用户的降级访问，所以 Web 应用可以查阅 Alice 的授权，并且让 Accumulo 的授权过滤器限制 Alice 对数据的授权访问。

11.5 Oozie

从客户端访问的角度看，Oozie 是个非常重要的工具。除了作为集群工作流的执行者和规划者之外，Oozie 还可以作为给客户端的网关服务提交任意类型的作业。这允许在屏蔽从客户端对 YARN 或 MR1 的直接访问的同时，仍能允许远程的作业提交。如果选择使用该类架构，那么为 Oozie 服务启用身份验证和授权机制会非常重要。5.2.5 节第 4 小节描述过如何配置 Kerberos 身份验证机制，6.5 节详述过其授权机制。

一旦在服务器端启用了那些特性参数，那么在客户端几乎不需要额外配置也能使用。要通过 Oozie 的身份验证，仅需要在自己的工作站上缓存 Kerberos 的 TGT。这很容易实现，只要在命令行执行 Oozie 命令之前就执行 kinit 即可。若执行了一个 Oozie 命令，并且看见了 `Failed to find any Kerberos tgt` 的错误消息，这说明很可能没有运行 `kinit`：

```
[alice@edge01 ~]$ oozie jobs -oozie http://oozie01.example.com:11000/oozie
Error: AUTHENTICATION : Could not authenticate, GSSException: No valid cre
ntials provided (Mechanism level: Failed to find any Kerberos tgt)
[alice@edge01 ~]$ klist
klist: No credentials cache found (ticket cache FILE:/tmp/krb5cc_123600000
[alice@edge01 ~]$ kinit
Password for alice@EXAMPLE.COM:
[alice@edge01 ~]$ oozie jobs -oozie http://oozie01.example.com:11000/oozie
No Jobs match your criteria!
```

我们已经配置了 Oozie 的身份验证和授权机制，但还没有做任何操作以保证 Oozie 客户端和服务器之间通信的机密性。幸运的是，Oozie 支持使用 HTTPS 加密连接，并且提供完整性检查。要启用 HTTPS，必须获取由证书颁发机构（CA）分发给 Oozie 服务的证书。关于如

何创建一个自签名证书的示例，详见 10.2.1 节。

一旦证书颁发机构分发了证书，且拥有证书和 PKCS12 格式的私钥，那么可以向 Java keystore 文件导入证书的私钥。下列示例中，给 keystore 和证书的私钥使用同样的通行码 `secret`：

```
[root@oozie01 ~]# mkdir /etc/oozie/ssl
[root@oozie01 ~]# keytool -v -importkeystore \
  -srckeystore /etc/pki/tls/private/oozie01.example.com.p12 \
  -srcstoretype PKCS12 \
  -destkeystore /etc/oozie/ssl/oozie01.example.com.keystore -deststoretype
  -deststorepass secret -srcalias oozie01.example.com -destkeypass secret
Enter source keystore password:
[Storing /etc/oozie/ssl/oozie01.example.com.keystore]
[root@oozie01 ~]# chown -R oozie:oozie /etc/oozie/ssl
[root@oozie01 ~]# chmod 400 /etc/oozie/ssl/*
[root@oozie01 ~]# chmod 700 /etc/oozie/ssl
```

接下来，在 `oozie-env.sh` 文件中设置控制 keystore 位置和口令的环境变量：

```
export OOZIE_HTTPS_KEYSTORE_FILE=/etc/oozie/ssl/oozie01.example.com.keysto
export OOZIE_HTTPS_KEYSTORE_PASS=secret
```

> 此处的 keystore 口令会对任何能够在运行 Oozie 的服务器上执行进程的人可见。必须对 keystore 文件采用强的权限保护措施，以防止用户读取或修改 keystore 文件。

配置 Oozie 使用 HTTPS 之前，需要确保 Oozie 服务没有运行。要配置 Oozie 使用 HTTPS，可以运行以下命令：

```
[oozie@oozie01 ~]$ oozie-setup.sh prepare-war -secure
```

现在，如果启用该服务，将会在端口 11443 上使用 HTTPS 协议。可以通过设置 `oozie-env.sh` 文件中的环境变量 `OOZIE_HTTPS_PORT` 改变端口。

在访问 Oozie 的客户端机器上，可以简单地通过命令行将 Oozie URL 改为 https://oozie01.example.com:11443/oozie，如：

```
[alice@edge01 ~]$ oozie jobs -oozie https://oozie01.example.com:11443/oozi
No Jobs match your criteria!
```

如果没有获得期望的输出，而得到了如下 SSLHandshakeException 错误消息：

```
[alice@edge01 ~]$ oozie jobs -oozie https://oozie01.example.com:11443/oozi
Error: IO_ERROR : javax.net.ssl.SSLHandshakeException: sun.security.valida
ValidatorException: PKIX path building failed: sun.security.provider.certp
SunCertPathBuilderException: unable to find valid certification path tore
target
```

这说明，Oozie 服务使用的证书没有被信任的证书颁发机构签名。这可能发生在使用自签名证书或某个内部 CA 没有被根 CA 签名的情况下。假设 example-ca.crt 文件中有一个 EXAMPLE.COM 域的证书颁发机构，由于要将该证书导入 Java 的 central truststore，所以该证书会被所有运行于该服务器上的 Java 应用所信任，而不仅仅是 Oozie。可以通过以下命令将其导入 Java 的 central truststore：

```
[root@edge01 ~]# keytool -import -alias EXAMPLE.COM -file example-ca.crt \
  -keystore ${JAVA_HOME}/jre/lib/security/cacerts
Enter keystore password:
Owner: CN=Certificate Authority, O=EXAMPLE.COM
Issuer: CN=Certificate Authority, O=EXAMPLE.COM
Serial number: 1
...
Trust this certificate? [no]: yes
Certificate was added to keystore
```

Java cacerts 文件的默认口令是 changeit。如果 Oozie 被配置为高可用性（HA），那么需要将负载均衡器配置进行 TLS（传输层安全）直连。这会允许客户端看见 Oozie 服务器的证书，且不需要负载均衡器有自己的证书。进行 TLS 直连操作时，应当使用通配证书或者包含负载均衡器全限定域名作为有效名称的证书。

11.6 Sqoop

第 10 章中，我们讨论了如何保护机密性、完整性和数据导入管道的可用性，这些原则也同样适用于保护数据提取管道。Sqoop 中，机密性不是由 Sqoop 本身提供的，但可以由 Sqoop 使用的、与 RDBMS 服务交互的驱动提供。10.2.2 节展示了应该怎样配置 MySQL 驱动，

以使用 SSL 对 MySQL 服务器和 Sqoop 执行的任务之间的通信进行加密。可以使用相同的参数配置加密那些正在导出的数据，如示例 11-7 所示。

示例 **11-7**　用 SSL 导出 MySQL 表

```
[alice@sqoop01 ~]$ hdfs dfs -cat cities/*
1,USA,Palo Alto
2,Czech Republic,Brno
3,USA,Sunnyvale
[alice@sqoop01 ~]$ URI="jdbc:mysql://mysql01.example.com/sqoop"
[alice@sqoop01 ~]$ URI="${URI}?verifyServerCertificate=false"
[alice@sqoop01 ~]$ URI="${URI}&useSSL=true"
[alice@sqoop01 ~]$ URI="${URI}&requireSSL=true"
[alice@sqoop01 ~]$ sqoop export --connect ${URI} \
  --username sqoop -P --table cities \
  --export-dir cities
Enter password:
...
14/06/28 17:27:22 INFO mapreduce.ExportJobBase: Exported 3 records.
[alice@sqoop01 ~]$
```

11.7　SQL访问

正如第 1 章所述，使用 SQL 访问 Hadoop 数据有两种流行的方法：Hive 和 Impala。这二者都支持 Kerberos 和基于 LDAP 的用户名 / 口令身份验证。用户通常不直接与 Hive 或 Impala 进行交互，而依赖 SQL shell 或 JDBC 驱动。

本章的剩余内容包括配置 Impala 和 Hive 支持的身份验证协议，以及作为客户端如何传递验证信息。Impala 和 Hive 的授权机制由 Sentry 提供，该部分已经在第 7 章讲过。

11.7.1　Impala

Impala 能被配置为仅使用 Kerberos、仅使用 LDAP 或同时使用 LDAP 和 Kerberos 进行身份验证。Impala 运行于启用 Kerberos 的 Hadoop 集群中时，必须被配置为使用 Kerberos，这样 Impala 才能与 HDFS 和 YARN 安全地进行通信。

01. 使用带有**Kerberos**验证机制的**Impala**

与 Hadoop 的通信启用了 Kerberos 时，基于 Kerberos 的客户

端身份验证机制会自动启用。Impala 使用命令行工具进行配置，应当对 impalad、statestored 和 catalogd 守护进程的 --principal 和 --keytab_file 参数进行配置。--principal 应被配置为 Impala 进行身份验证的 Kerberos 主体，通常格式是 impala/<fully qualified domain name>@<realm>，其中 <fully qualified domain name> 是运行着 impalad 的主机名，<realm> 是 Kerberos 域。主体的第一个组件 impala 必须与启动 Impala 进程的用户名相匹配。--keytab_file 参数必须指向一个包含前述主体和运行有 impalad 服务器的 HTTP 主体的 keytab 文件。可以借助 ktutil 命令，用来自两个独立 keytab 的文件创建一个 keytab 文件，如示例 11-8 所示。

示例 **11-8**　融合 Impala 和 HTTP 的 keytab

```
[impala@impala01 ~]$ ktutil
ktutil: rkt impala.keytab
ktutil: rkt http.keytab
ktutil: wkt impala-http.keytab
ktutil: quit
```

为了使配置更容易，可以对 impalad 进程使用的命令行参数进行设置，设置 /etc/default/impala 文件中的 IMPALA_SERVER_ARGS、IMPALA_STATE_STORE_ARGS 和 IMPALA_CATALOG_ARGS 变量即可。

示例 **11-9**　为 Impala 配置 Kerberos 身份验证

```
IMPALA_SERVER_ARGS="${IMPALA_SERVER_ARGS} \
  --principal=impala/impala01.example.com@EXAMPLE.COM \
  --keytab_file=/etc/impala/conf/impala-http.keytab"

IMPALA_STATE_STORE_ARGS="${IMPALA_STATE_STORE_ARGS} \
  --principal=impala/impala01.example.com@EXAMPLE.COM \
  --keytab_file=/etc/impala/conf/impala-http.keytab"

IMPALA_CATALOG_ARGS="${IMPALA_CATALOG_ARGS} \
  --principal=impala/impala01.example.com@EXAMPLE.COM \
  --keytab_file=/etc/impala/conf/impala-http.keytab"
```

如果用户在负载均衡之后访问 Impala，那么配置需要一些小小

的改动。创建融合的 keytab 文件时，还需要加入代理服务主体的 keytab，并且添加 `--be_principal` 参数。`--be_principal` 参数是 Impala 与诸如 HDFS 等后端服务进行通信的主体，其值应与之前设置的 `--principal` 参数一样，且 `--principal` 参数应被改为负载均衡的主体。若负载均衡器在 `impala-proxy.example.com` 上，那么应当如示例 11-10 所示设置 `IMPALA_SERVER_ARGS`。

示例 **11-10** 配置负载均衡器之后的 Impala 的 Kerberos 验证机制

```
IMPALA_SERVER_ARGS="${IMPALA_SERVER_ARGS} \
  --principal=impala/impala-proxy.example.com@EXAMPLE.COM \
  --be_principal=impala/impala01.example.com@EXAMPLE.COM \
  --keytab_file=/etc/impala/conf/impala-http.keytab"
```

Impala 支持用户通过 Llama 项目使用 YARN，以进行资源管理。Llama 能协调 YARN 和 Impala 这种低时延执行引擎之间的资源管理。Llama 有两个组件：一个长期运行的应用控制程序，一个节点管理插件。应用控制程序处理 Impala 的保留资源，节点管理插件与本地 Impala 守护程序合作调节本地节点的可用资源。

Impala 启用 Kerberos 时，必须对 `llama-site.xml` 文件中的下列参数进行设置，以配置 Llama 的 Kerberos。

`lama.am.server.thrift.security`

将其设为 `true`，以便为应用控制程序启用 Thrift SASL/基于 Kerberos 的安全策略。

`llama.am.server.thrift.security.QOP`

启用安全策略时，用其设置保护级别。有效参数为：仅进行身份验证设为 `auth`，进行身份验证和完整性检查设为 `auth-int`，进行身份验证、完整性检查和机密性保障（加密）则设为 `auth-conf`。

`llama.am.server.thrift.kerberos.server.principal.na`

Llama 应用服务器的完全限定主体名。该配置必须同时包

含运行有 Llama 应用控制程序的服务器的短名称和完全限定主机名。

`llama.am.server.thrift.kerberos.notification.princi`

用于客户端通知的短名称。该短名称结合了注册期间 impalad 进程提供的客户端主机名。可以通过配置 `IMPALA_SERVER_ARGS` 变量中的 `--hostname` 参数覆盖该主机名。

示例 11-11 展示了 `llama-site.xml` 文件配置启用 Kerberos 的片段。

示例 **11-11**　配置 Llama 应用控制程序的 Kerberos 验证机制

```
<property>
  <name>llama.am.server.thrift.security</name>
  <value>true</value>
</property>
<property>
  <name>llama.am.server.thrift.kerberos.keytab.file</name>
  <value>/etc/llama/conf/llama.keytab</value>
</property>
<property>
  <name>llama.am.server.thrift.kerberos.server.principal.name</name>
  <value>llama/llama.example.com@EXAMPLE.COM</value>
</property>
<property>
  <name>llama.am.server.thrift.kerberos.notification.principal.name<
  <value>impala</value>
</property>
```

一旦 Impala 被配置为使用 Kerberos 验证机制，客户端就可以通过缓存的 Kerberos TGT 进行身份验证（如在执行控制台之前运行 `kinit`）。示例 11-12 展示了 Alice 获取其 Kerberos TGT 并使用 Impala 控制台进行 Kerberos 身份验证的过程。

示例 **11-12**　使用 Impala 控制台进行 Kerberos 身份验证

```
[alice@hadoop01 ~]$ kinit
Password for alice@EXAMPLE.COM:
[alice@hadoop01 ~]$ impala-shell -i impala-proxy
Starting Impala Shell without Kerberos authentication
```

```
Error connecting: TTransportException, TSocket read 0 bytes
Kerberos ticket found in the credentials cache, retrying the connect
ecure transport.
Connected to impala-proxy:21000
Server version: impalad version 2.0.0 RELEASE (build ecf30af0b4d6e56
9367ada6b7da7)
Welcome to the Impala shell. Press TAB twice to see a list of availa

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v2.0.0 (ecf30af) built on Sat Oct
 PDT 2014)
[impala-proxy:21000] > show tables;
Query: show tables
+-----------+
| name      |
+-----------+
| sample_07 |
| sample_08 |
+-----------+
Fetched 2 row(s) in 0.18s
[impala-proxy:21000] >
```

通过 JDBC 驱动进行 Kerberos 身份验证时，首先需要获取一个 Kerberos TGT，然后将 Impala 主体的名称包含在连接字符串中。示例 11-13 展示了如何为 Impala 的 Kerberos 验证设置 JDBC 连接字符串。

示例 **11-13**　用于 Kerberos 身份验证的 JDBC 连接字符串

```
//首先定义一个基本的JDBC URL字符串
String url = "jdbc:hive2://impala-proxy.example.com:21050/default";

//添加Impala Kerberos主体名，包括FQDN和域
url = url + ";principal=impala/impala-proxy.example.com@EXAMPLE.COM"

// 使用URL创建连接
Connection con = DriverManager.getConnection(url);
```

02. 使用带有**LDAP/ Active Directory**验证的**Impala**

Impala 还可以被配置为使用 LDAP 目录进行身份验证，例如 Active Directory。使用 LDAP 验证的优势在于，其客户端不需要在连接到 Impala 之前先获取它们的 Kerberos 证书。这个优势对于那些可能在本地不支持基于 Kerberos 验证的商业智能软

件来说，是很有帮助的。虽然 LDAP 身份验证的配置独立于 Kerberos，但二者通常会被一起配置，因为 Hadoop 没有启用身份验证机制时，只单独给 Impala 启动身份验证机制也没有多大意义。

设置 impalad 守护进程中的 --enable_ldap_auth 和 --ldap_uri parameters 参数，启用 LDAP 身份验证机制。配置 LDAP 时，可以根据使用的 LDAP 的提供者选择性地设置绑定参数。若使用 Active Directory，则通常不需要进行额外的配置，但需要明确设置域名，从而使与 AD（Active Directory）绑定的用户名以 user@<domain name> 的形式传递。该域名通过参数 --ldap_domain 进行设置。对于 OpenLDAP 或 freeIPA，可以配置一个基准标识域名，从而使与 LDAP 绑定的用户名以 uid=user,<base dn> 的形式传递。该设置通过参数 --ldap_baseDN 启用。若 LDAP 提供者不使用 uid=user 指定用户名中的唯一性名称，则可以提供一个会被当成唯一性名称的格式。通过用首选绑定的用户名取代所有的 #UID 实例得到实现。该设置通过配置参数 --ldap_bind_pattern 启用。

不考虑 LDAP 提供者，强烈推荐使用 TLS 加密 Impala 和 LDAP 服务器之间的连接，可以通过一个 ldaps://URL 或启用 StartTLS 完成。可以将参数 --ldap_tls 设为 true 以启用 StartTLS。对于任一种模式，必须配置证书颁发机构（CA）的证书，以使 Impala 能够信任 LDAP 服务器的证书。可以通过设置参数 --ldap_ca_certificate 对 CA 证书的位置进行配置。

示例 11-14~示例 11-16 分别使用 Active Directory、OpenLDAP 和自定义 LDAP 提供者进行配置。

示例 **11-14**　配置使用 Active Directory 的 Impala

```
IMPALA_SERVER_ARGS="${IMPALA_SERVER_ARGS} \
  --enable_ldap_auth=true \
  --ldap_uri=ldaps://ad.example.com \
  --ldap_ca_certificate=/etc/impala/pki/ca.crt \
  --ldap_domain=example.com"
```

示例 **11-15**　配置使用 OpenLDAP 的 Impala

```
IMPALA_SERVER_ARGS="${IMPALA_SERVER_ARGS} \
  --enable_ldap_auth=true \
  --ldap_uri=ldaps://ldap.example.com \
  --ldap_ca_certificate=/etc/impala/pki/ca.crt \
  --ldap_baseDN=ou=People,dc=example,dc=com"
```

示例 **11-16**　配置使用其他 LDAP 的 Impala

```
IMPALA_SERVER_ARGS="${IMPALA_SERVER_ARGS} \
  --enable_ldap_auth=true \
  --ldap_uri=ldaps://ldap.example.com \
  --ldap_ca_certificate=/etc/impala/pki/ca.crt \
  --ldap_bind_pattern=user=#UID,ou=users,dc=example,dc=com"
```

要使用 LDAP 验证机制连接 Impala，将命令行的 `-l` 选项配置为 `impala-shell`，连接建立完成前会要求键入用户口令。如果想用与当前 Linux 用户不同的另一个用户身份进行连接，可以使用 `-u` 选项更改用户名。示例 11-17 展示了如何通过 Impala 控制台使用 LDAP 进行身份验证。

示例 **11-17**　使用 LDAP/Active Directory 身份验证的 Impala 控制台

```
[alice@hadoop01 ~]$ impala-shell -i impala-proxy -l
Starting Impala Shell using LDAP-based authentication
LDAP password for alice:
Connected to impala-proxy:21000
Server version: impalad version 2.0.0 RELEASE (build ecf30af0b4d6e56
f2189367ada6b7da7)
Welcome to the Impala shell. Press TAB twice to see a list of availa

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v2.0.0 (ecf30af) built on Sat Oct
6:06 PDT 2014)
[impala-proxy:21000] > show tables;
Query: show tables
+-----------+
| name      |
+-----------+
| sample_07 |
| sample_08 |
+-----------+
Fetched 2 row(s) in 0.16s
[impala-proxy:21000] >
```

如果使用 JDBC 驱动连接 Impala，则需要在建立连接时将用户名和口令传递给 DriverManager。示例 11-18 展示了如何使用 JDBC 驱动连接使用 LDAP 验证的 Impala。

示例 **11-18**　用于 LDAP/Active Directory 身份验证的 KDBC 连接串

```
// 定义一个基本的JDBC URL字符串
String url = "jdbc:hive2://impala-proxy.example.com:21050/default";

// 使用URL、用户名和密码创建连接
Connection con = DriverManager.getConnection(url, "alice", "secret");
```

03. 为**Impala**启用**SSL**传输加密

之前介绍的方法涵盖了从客户端进行身份验证的不同途径。为客户端和 Impala 之间的数据传输建立一个受保护的通道也很重要，Impala 处理的数据较为敏感时，这更重要——例如数据需要被静态加密时。出于这些目的，Impala 支持 SSL 传输加密。示例 11-19 展示了必需的启动标识。

示例 **11-19**　对 Impala 进行 SSL 配置

```
IMPALA_SERVER_ARGS="${IMPALA_SERVER_ARGS} \
  --ssl_client_ca_certificate=/etc/impala/ca.cer \
  --ssl_private_key=/etc/impala/impala.key \
  --ssl_server_certificate=/etc/impala/impala.cer
```

`ssl_private_key`、`ssl_server_certificate` 和 `ssl_client_ca_certificate` 的路径对于 `Impala` 用户来说都必须是可读的，并且证书必须是 PEM 格式。推荐将私钥的权限限制为 400。

Impala 配置了 SSL 时，客户端还必须知道如何正确连接服务端。`--ssl` 选项能够让 `Impala-shell` 为连接而启用 SSL，`--ca_cert` 参数能指定用于验证连接的 Impala 守护程序提供证书的 CA 链（以 PEM 的格式）。示例 11-20 展示了同时使用 Kerberos 验证机制和 SSL 传输加密的情况。

示例 **11-20**　启用 SSL 和 Kerberos 的 Impala 控制台

```
alice@hadoop01 ~]$ impala-shell -i impala-proxy -k --ssl --ca_cert /
Starting Impala Shell using Kerberos authentication
SSL is enabled
Connected to impala-proxy:21000
Server version: impalad version 2.0.0 RELEASE (build ecf30af0b4d6e56
f2189367ada6b7da7)
Welcome to the Impala shell. Press TAB twice to see a list of availa

Copyright (c) 2012 Cloudera, Inc. All rights reserved.

(Shell build version: Impala Shell v2.0.0 (ecf30af) built on Sat Oct
6:06 PDT 2014)
[impala-proxy:21000] > show tables;
Query: show tables
+-----------+
| name      |
+-----------+
| sample_07 |
| sample_08 |
+-----------+
Fetched 2 row(s) in 0.16s
[impala-proxy:21000] >
```

11.7.2 Hive

陈旧过时的 Hive 命令行工具 hive 并不能支持直接的 Hive 身份验证或授权，而是要么直接访问 HDFS 上的数据，要么运行一个 MapReduce 作业执行查询。这意味着它与我们之前描述的 Hadoop 命令遵循着同样的规则，只支持 Kerberos 和委托令牌。`hive` 命令基本已被弃用，用户应当使用 `beeline`。

使用 `beeline` 或 JDBC 驱动时，用户要连接到负责解析查询和执行的 HiveServer2 服务进程。HiveServer2 支持 Kerberos、LDAP 和自定义身份验证插件。由于一次只能配置一个认证提供者（authentication provider），所以管理员配置 HiveServer2 服务进程时，需要选择其首选的身份验证机制。解决这种限制的一个方法是，运行多个共享同一 Hive 元数据的 HiveServer2 守护进程。这要求终端用户能够根据其身份验证需求连接到正确的 HiveServer2。HiveServer2 的身份验证机制可以在 `hive-site.xml` 文件中配置。HiveServer2 身份验证配置属性的描述见表 11-2。

表11-2：配置HiveServer2身份验证属性

属性	描述

hive.server2.authentication	客户认证类型，有效值包括：NONE、LDAP、KERBEROS、CUSTOM
hive.server2.authentication.kerberos.principal	HiveServer2 服务进程的 Kerberos 主体
hive.server2.authentication.kerberos.keytab	用于 KDC 身份验证的 keytab
hive.server2.thrift.sasl.qop	Kerberos 连接使用的 SASL 质量保护；有效值包括：auth 表示仅身份验证；auth-int 表示身份验证和完整性验证；auth-conf 表示身份验证、完整性和机密性（加密）
hive.server2.use.SSL	设置为 true，在客户端与 HiveServer2 服务之间启用 TLS
hive.server2.keystore.path	包含 TLS 要使用的私钥的 Java keystore 文件路径
hive.server2.keystore.password	Java keystore 文件的密码
hive.server2.authentication.ldap.url	LDAP/Active Directory 服务器的 URL；仅当 hive.server2.authentication 设为 LDAP 时使用
hive.server2.authentication.ldap.Domain	进行身份验证的Active Directory 域；仅当 hive.server2.authentication.ldap.url 指向 AD 服务器时使用
hive.server2.authentication.ldap.baseDN	指向 OpenLDAP 服务器时使用的基准标识名
hive.server2.custom.authentication.class	实现 org.apache.hive.service.auth.PasswdAuthenticationProviderinterface的类名，hive.server2.authenticationis 设为 CUSTOM 时使用

01. 使用**HiveServer2**的**Kerberos**身份验证

配置 HiveServer2 的 Kerberos 身份验证与 5.2.5 节描述的配置其他核心 Hadoop 服务的模式相同。也就是说，需要将认证类型设为 Kerberos，并且设置 Kerberos 主体和 keytab。设置 Kerberos 主体时，可以使用 _HOST 占位符，它会被自动替换为运行 HiveServer2 服务器的完全限定域名。示例 11-21 展示了一个启用 Kerberos 身份验证的 hive-site.xml 示例片段。

示例 **11-21**　配置 HiveServer2 的 Kerberos 身份验证

```
<property>
  <name>hive.server2.authentication</name>
   <value>KERBEROS</value>
</property>
<property>
  <name>hive.server2.authentication.kerberos.principal</name>
  <value>hive/_HOST@EXAMPLE.COM</value>
</property>
<property>
  <name>hive.server2.authentication.kerberos.keytab</name>
  <value>/etc/hive/conf/hive.keytab</value>
</property>
```

要想连接到启用了 Kerberos 的 HiveServer2 服务进程的 JDBC 客户端，需要拥有一个有效的 Kerberos TGT，并将 HiveServer2 服务进程的主体添加到连接字符串。如示例 11-22

所示，创建一个用于 Kerberos 身份验证的连接字符串。

示例 **11-22** 用于 Kerberos 身份验证的 JDBC 连接字符串

```
//首先定义一个基本JDBC URL字符串
String url = "jdbc:hive2://hive.example.com:10000/default";

// 添加Hive Kerberos主体名，包括FQDN和域
url = url + ";principal=hive/hive.example.com@EXAMPLE.COM";

// 根据URL创建连接
Connection con = DriverManager.getConnection(url);
```

Beeline 控制台使用 Hive JDBC 驱动连接 HiveServer2。需要使用 kinit 获取 Kerberos TGT，然后使用 JDBC 连接字符串在 Beeline 中进行 Kerberos 身份验证，如示例 11-23 所示。即使在使用 Kerberos 进行身份验证，Beeline 依然会提示输入用户名和密码。可以将这些内容留空，直接敲回车。

示例 **11-23** 用于 Kerberos 身份验证的 Beeline 连接字符串

```
[alice@hadoop01 ~]$ kinit
Password for alice@EXAMPLE.COM:
[alice@hadoop01 ~]$ beeline
Beeline version 0.13.1 by Apache Hive
beeline> !connect jdbc:hive2://hive.example.com:10000/default;princ
e.example.com@EXAMPLE.COM
scan complete in 2ms
Connecting to jdbc:hive2://hive.example.com:10000/default;principal=
ample.com@EXAMPLE.COM
Enter username for jdbc:hive2://hive.example.com:10000/default;princ
ve.example.com@EXAMPLE.COM:
Enter password for jdbc:hive2://hive.example.com:10000/default;princ
ve.example.com@EXAMPLE.COM:
Connected to: Apache Hive (version 0.13.1)
Driver: Hive JDBC (version 0.13.1)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://hive.example.com> show tables;
+------------+--+
|  tab_name  |
+------------+--+
| sample_07  |
| sample_08  |
+------------+--+
2 rows selected (0.261 seconds)
```

```
0: jdbc:hive2://hive.example.com>
```

02. 使用HiveServer2的LDAP/ Active Directory身份验证

HiveServer2 也支持基于 LDAP 的用户名 / 口令身份验证。要使用基于 LDAP 的身份验证，需要将认证类型设为 LDAP，配置 LDAP URL，并且之后设置一个用于绑定的域名或基准标识名（base distinguished name）。绑定 Active Directory 服务器时使用域名，而绑定 OpenLDAP 或 freeIPA 等其他 LDAP 提供者时，使用基准标识名。

> 默认情况下，客户端和 HiveServer2 的连接是没有加密的。这意味着，使用 LDAP 或自定义认证提供者时，用户名和口令都有可能被第三方拦截。使用非 Kerberos 认证提供者时，强烈建议使用 TLS 开启 HiveServer2 的线上加密，详情参见后文。

无论是哪种 LDAP 提供者，我们都强烈建议不要直接使用 LDAP，而使用 LDAPS（LDAP over SSL），这能够保证 HiveServer2 和 LDAP 服务器之间的通信是加密的。要使用 LDAPS，需要确保 LDAP 服务器证书或者 CA 签名证书加载到 Java 信任库（truststore）。可以是位于 `$JAVA_HOME/jre/lib/security/cacerts` 的整个系统的 Java 信任库，也可以是用于 Hive 的特定信任库。如果使用特定的信任库，需要设置 `javax.net.ssl.trustStore` 和 `javax.net.ssl.trustStorePassword` 系统属性。这可以通过设置 `hive-env.sh` 文件中的 `HADOOP_OPTS` 变量实现，如示例 11-24 所示。

示例 **11-24**　为 Hive 设置 LDAPS 信任库

```
HADOOP_OPTS="-Djavax.net.ssl.trustStore=/etc/pki/java/hive.truststor
HADOOP_OPTS="${HADOOP_OPTS} -Djavax.net.ssl.trustStorePassword=secre
```

> 此处使用的信任库密码对能够在运行 HiveServer2 的服务器上枚举进程的任意用户都是可见的。必须用强制权限保护信任库文件本身，以防其他用户修改信任库。

如果要配置 HiveServer2，以对 Active Directory 服务器进行身份验证，那么除了常规的 LDAP 设置之外，还需要将 `hive-site.xml` 中的 `hive.server2.authentication.ldap.Domain` 设置为 AD 域名，如示例 11-25 所示。

示例 11-25 配置 HiveServer2 中的 Active Directory 身份验证

```
<property>
  <name>hive.server2.authentication</name>
  <value>LDAP</value>
</property>
<property>
  <name>hive.server2.authentication.ldap.url</name>
  <value>ldaps://ad.example.com</value>
</property>
<property>
  <name>hive.server2.authentication.ldap.Domain</name>
  <value>example.com</value>
</property>
```

如果使用的是其他 LDAP 提供者，如 OpenLDAP、freeIPA 等，那么需要设置 `hive.server2.authentication.ldap.baseDN` 属性，而不是设置域名。基准 DN 取决于环境，但通常 OpenLDAP 安装的默认值是 `ou=People,dc=example,dc=com`，其中 `dc=example,dc=com` 要替换为 LDAP 服务器域组件，这通常是 LDAP 服务器的域名。对于 freeIPA，默认的基本 DN 是 `cn=users,cn=accounts,dc=example,dc=com`，同样，也要根据环境替换其中的域组件。示例 11-26 提供了完整配置。

示例 11-26 配置 HiveServer2 中的 LDAP 身份验证

```
<property>
  <name>hive.server2.authentication</name>
  <value>LDAP</value>
</property>
<property>
  <name>hive.server2.authentication.ldap.url</name>
  <value>ldaps://ldap.example.com</value>
</property>
```

```
<property>
  <name>hive.server2.authentication.ldap.baseDN</name>
  <value>ou=People,dc=example,dc=com</value>
</property>
```

某些版本的 Hive（尤其是 Hive 0.13.0 和 0.13.1）存在一个 bug，即身份验证类型被设为除 KERBEROS 之外的其他值时，它们就不会使用 Kerberos 身份验证与 Hadoop 通信。使用这些版本的 Hive 时，应当只使用 Kerberos 进行身份验证。

配置 LDAP/Active Directory 身份验证后，连接 HiveServer2 就很容易了。只需在建立连接之后，将用户名和口令传递给 DriverManager 即可，如示例 11-27 所示。

示例 **11-27** LDAP/Active Directory 身份验证的 JDBC 连接字符串

```
// 使用基本JDBC URL字符串
String url = "jdbc:hive2://hive.example.com:10000/default";

// 根据URL创建提交用户名和密码的连接
Connection con = DriverManager.getConnection(url, "alice", "secret")
```

连接 Beeline 的方法基本相同。这次要在 !connect 命令后、出现提示时，输入用户名和口令，如示例 11-28 所示。

示例 **11-28** LDAP/Active Directory 身份验证的 Beeline 连接字符串

```
[alice@hadoop01 ~]$ beeline
Beeline version 0.13.1 by Apache Hive
beeline> !connect jdbc:hive2://hive.example.com:10000/default
scan complete in 2ms
Connecting to jdbc:hive2://hive.example.com:10000/default
Enter username for jdbc:hive2://hive.example.com:10000/default: alic
Enter password for jdbc:hive2://hive.example.com:10000/default: ****
Connected to: Apache Hive (version 0.13.1)
Driver: Hive JDBC (version 0.13.1)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://hive.example.com> show tables;
+------------+--+
|  tab_name  |
```

```
+------------+--+
| sample_07  |
| sample_08  |
+------------+--+
2 rows selected (0.261 seconds)
0: jdbc:hive2://hive.example.com>
```

03. 使用**HiveServer2**的可插入身份验证

Hive 具有实现新认证提供者的可插入接口，Hive 将这种身份验证模式称为 `CUSTOM`，它需要一个实现了 `org.apache.hive.service.auth.PasswdAuthenticationPr` 接口的 Java 类。该接口定义了一个 `authenticate(String user, String password)` 方法，应当实现该方法以验证用户提交的用户名和密码。顾名思义，该可插入接口只支持使用用户名和密码进行身份验证的认证提供者。要配置这种身份验证模式，需要将认证类型设为 `CUSTOM`，并配置好身份验证类，还需要将包含身份验证类的 JAR 文件添加到 Hive 的 classpath。最简单的方法是，将 JAR 文件路径添加到 `hive-site.xml` 中的 `hive.aux.jars.path` 配置项，该配置项接受的输入为逗号分隔的 JAR 文件全路径（示例 11-29）。

示例 **11-29** 配置 HiveServer2 中的可插入身份验证

```
<property>
  <name>hive.server2.authentication</name>
  <value>CUSTOM</value>
</property>
<property>
  <name>hive.server2.custom.authentication.class</name>
  <value>com.example.my.whizbang.AuthenticationProvider</value>
</property>
<property>
  <name>hive.aux.jars.path</name>
  <value>file:///opt/hive-plugins/whizbang-1.0.jar</value>
</property>
```

自定义身份验证的连接设置与基于 LDAP/Active Directory 的身份验证一样（示例 11-18）。

04. **HiveServer2**线上加密

Hive JDBC 驱动支持两种启用线上加密的方法，使用哪种方法取决于使用的认证方法和 Hive 版本。使用 Kerberos 认证时，Hive JDBC 驱动使用 SASL 执行 Kerberos 认证。SASL 在基于一个名为保护质量（**quality of protection**，**QOP**）的配置项进行身份验证时，支持完整性验证和加密。要启用加密，需要将 SASL QOP 设置为 `auth-conf`，它是保密性身份验证（authentication with confidentiality）的简称。示例 11-30 展示了如何配置 HiveServer2 以使用 SASL 进行加密。

示例 **11-30**　配置 HiveServer2 使用 SASL 加密

```
<property>
  <name>hive.server2.thrift.sasl.qop</name>
  <value>auth-conf</value>
</property>
```

服务端启用 SASL QOP 后，需要确保客户端也将其设置成同样的值。可以将 `sasl.qop=auth-conf` 选项添加到 JDBC URL 以实现。示例 11-31 展示了如何在 Beeline 中使用 SASL 加密。

示例 **11-31**　使用 SASL 加密的 Beeline 连接字符串

```
[alice@hadoop01 ~]$ beeline
Beeline version 0.13.1 by Apache Hive
beeline> !connect jdbc:hive2://hive.example.com:10000/default;princ
e.example.com@EXAMPLE.COM;sasl.qop=auth-conf
scan complete in 4ms
Connecting to jdbc:hive2://hive.example.com:10000/default;principal=
ample.com@EXAMPLE.COM;sasl.qop=auth-conf
Enter username for jdbc:hive2://hive.example.com:10000/default;princ
ve.example.com@EXAMPLE.COM;sasl.qop=auth-conf:
Enter password for jdbc:hive2://hive.example.com:10000/default;princ
ve.example.com@EXAMPLE.COM;sasl.qop=auth-conf:
Connected to: Apache Hive (version 0.13.1)
Driver: Hive JDBC (version 0.13.1)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://hive.example.com> show tables;
+------------+--+
|  tab_name  |
+------------+--+
| sample_07  |
| sample_08  |
```

```
+------------+--+
2 rows selected (0.261 seconds)
0: jdbc:hive2://hive.example.com>
```

如果已经配置 Hive 使用基于用户名 / 密码的身份验证，如 LDAP/Active Directory，那么 Hive 就不会再使用 SASL 进行安全连接了。这意味着需要另外一种方法启用加密。从 Hive 0.13 开始，可以配置 Hive 0.13 或更新的版本，以使用 TLS/SSL 进行加密。配置 Hive 使用 TLS 之前，需要将服务器的私钥和证书放入一个 Java keystore 文件。假定已经有了一个包含私钥和证书的 PKCS12 文件，则可以按照示例 11-32 所示的步骤将其导入 Java keystore。 Hive 要求私钥的口令必须设置成与 keystore 的口令相同的值。示例中，通过在命令行设置 `-deststorepass` 和 `-destkeypass` 满足这个要求。此外，还为要导入的密钥 / 证书提供 `-srcalias` 参数。

关于 Hive 版本的注意事项

仅 Hive 0.12.0 或更新版本能够设置 SASL QOP 属性，对 TLS 加密的支持则要求 Hive 0.13.0 或更新版本。

示例 11-32 导入 PKCS12 私钥到 Java keystore

```
[root@hive ~]# mkdir /etc/hive/ssl
[root@hive ~]# keytool -v -importkeystore \
  -srckeystore /etc/pki/tls/private/hive.example.com.p12 -srcstorety
  -destkeystore /etc/hive/ssl/hive.example.com.keystore -deststorety
  -deststorepass secret -srcalias hive.example.com -destkeypass secr
Enter source keystore password:
[Storing /etc/hive/ssl/hive.example.com.keystore]
[root@hive ~]# chown -R hive:hive /etc/hive/ssl
[root@hive ~]# chmod 400 /etc/hive/ssl/*
[root@hive ~]# chmod 700 /etc/hive/ssl
```

创建 Java keystore 之后，可以配置 Hive。如示例 11-33 所示，设置 `hive-site.xml` 文件中的配置项。

示例 11-33 配置 HiveServer2 以使用 TLS 进

行加密

```
<property>
  <name>hive.server2.use.SSL</name>
  <value>true</value>
</property>
<property>
  <name>hive.server2.keystore.path</name>
  <value>/etc/hive/ssl/hive.example.com.keystore</value>
</property>
<property>
  <name>hive.server2.keystore.password</name>
  <value>secret</value>
</property>
```

使用 Kerberos 进行身份验证时，无法配置 TLS。如果在使用 Kerberos 进行身份验证，那么可以使用 SASL QOP 进行加密，否则使用 TLS。

最后，需要在客户端将 `ssl=true` 添加到 JDBC URL，以启用 TLS。如果证书不是由中央认证中心签发的，则还需要在 JDBC URL 中指定一个信任库。配置 Hive 使用 LDAPS 时，我们曾创建一个信任库，此处可以通过复制信任库文件到客户端服务器，并设置 JDBC URL 中的 `sslTrustStore` 和 `trustStorePassword` 参数复用该信任库。示例 11-34 在 Beeline 中使用 TLS 进行加密。

示例 **11-34**　使用 TLS 的 Beeline 连接字符串

```
[alice@hadoop01 ~]$ beeline
Beeline version 0.13.1 by Apache Hive
beeline> !connect jdbc:hive2://hive.example.com:10000/default;ssl=tr
tore=/etc/pki/java/hive.truststore;trustStorePassword=secret
scan complete in 3ms
Connecting to jdbc:hive2://hive.example.com:10000/default;ssl=true;s
=/etc/pki/java/hive.truststore;trustStorePassword=secret
Enter username for jdbc:hive2://hive.example.com:10000/default;ssl=t
Store=/etc/pki/java/hive.truststore;trustStorePassword=secret alice
Enter password for jdbc:hive2://hive.example.com:10000/default;ssl=t
Store=/etc/pki/java/hive.truststore;trustStorePassword=secret *****
Connected to: Apache Hive (version 0.13.1)
Driver: Hive JDBC (version 0.13.1)
Transaction isolation: TRANSACTION_REPEATABLE_READ
0: jdbc:hive2://hive.example.com> show tables;
+------------+--+
```

```
|  tab_name  |
+------------+--+
| sample_07  |
| sample_08  |
+------------+--+
2 rows selected (0.261 seconds)
0: jdbc:hive2://hive.example.com>
```

11.8　WebHDFS/HttpFS

Hadoop 有两种方法对 HDFS 开放 REST 接口：WebHDFS 和 HttpFS。两种系统都使用了同样的 API，因此同样的客户端能够使用这两种中的任何一种，区别在于部署方式以及数据的访问方式。WebHDFS 实际上并不是个独立的服务，它运行在 NameNode 和 DataNode 上，不适合不能直接访问集群的用户。实际上，WebHDFS 最常被用来提供版本独立的批量访问功能，例如分布式复制命令 DistCp。示例 5-10 给出了 WebHDFS 的配置示例。

相比之下，HttpFS 作为一个网关服务运行，类似 HBase REST 网关。配置 HttpFS 身份验证的第一步是配置 HttpFS，以使用 Kerberos 对 HDFS 进行身份验证。

```
<property>
  <name>httpfs.hadoop.authentication.type</name>
  <value>kerberos</value>
</property>
<property>
  <name>httpfs.hadoop.authentication.kerberos.principal</name>
  <value>httpfs/httpfs.example.com@EXAMPLE.COM</value>
</property>
<property>
  <name>httpfs.hadoop.authentication.kerberos.keytab</name>
  <value>httpfs.keytab</value>
</property>
```

接下来，设置 HttpFS 服务器验证客户端身份的方式。依然使用 Kerberos，这将在 SPNEGO 上实现。

```
<property>
  <name>httpfs.authentication.type</name>
  <value>kerberos</value>
</property>
<property>
```

```
  <name>httpfs.authentication.kerberos.principal</name>
  <value>HTTP/httpfs.example.com@EXAMPLE.COM</value>
</property>
<property>
  <name>httpfs.authentication.kerberos.keytab</name>
  <value>httpfs.keytab</value>
</property>
<property>
  <name>httpfs.authentication.kerberos.name.rules</name>
  <value>DEFAULT</value>
</property>
```

最后，需要配置 HttpFS 允许 Hue 用户模拟其他用户的身份，这可以通过典型的代理用户设置实现。例如，下列配置将允许 `hue` 用户模拟来自任意主机任意组的用户。

```
<property>
  <name>httpfs.proxyuser.hue.hosts</name>
  <value>*</value>
</property>
<property>
  <name>httpfs.proxyuser.hue.groups</name>
  <value>*</value>
</property>
```

11.9　小结

本章深入介绍了客户端如何访问 Hadoop 集群，从而使用其提供的众多服务和存储的数据。显而易见，由于提供给客户端的访问点数量众多，保证这种访问的安全是非常艰巨的任务。但有一个关键点始终贯穿其中：客户端必须服从已制定的认证和授权方法，例如 Kerberos 和 LDAP 提供的。

本章还讲述了用户如何使用 Sqoop、Hive、Impala、WebHDFS 和 HttpFS 从集群获取数据。虽然 Hadoop 生态系统本身已经发展多年，但商业智能的庞大生态系统 ETL 也有相应发展，其他与 Hadoop 交互的相关工具亦是如此。因此，对于管理员而言，牢牢掌握平台的数据提取功能以及保护其安全的模式是至关重要的。

第 12 章　Cloudera Hue

Hue 是一个 Web 应用，它为很多 Hadoop 生态系统项目提供侧重于终端用户的接口。Hadoop 配置了 Kerberos 认证后，Hue 也必须配置 Kerberos 证书，以正常访问 Hadoop。用户可以通过配置 `hue.ini` 文件中的如下参数启用 Kerberos。

`hue_principal`

Hue 的 Kerberos 主体名称，包括 Hue 服务器的完全限定域名。

`hue_keytab`

包含 Hue 的服务证书的 Kerberos keytab 文件路径。

`kinit_path`

Kerberos `kinit` 命令的路径。

`reinit_frequency`

Hue 更新其 Kerberos 票据的频率（秒）。

这些配置应当放在 `hue.ini` 文件顶层 `[desktop]` 部分的 `[[kerberos]]` 小节。Hue kerberos 配置如示例 12-1 所示。

示例 12-1　配置 Hue 中的 Kerberos

```
[desktop]
[[kerberos]]
hue_principal=hue/hue.example.com@EXAMPLE.COM
hue_keytab=/etc/hue/conf/hue.keytab
reinit_frequency=3600
```

Hue 有自己的一套认证后端，能够对 Hadoop 和其他使用 Kerberos 的项目进行身份验证。为了代表其他用户执行动作，Hadoop 必须配置为信任 Hue 服务。这可以通过配置 Hadoop 的代理用户／用户模拟功能实现。需要对 Hue 可以运行的主机和 Hue 可以模拟的用户组进行设置，任何一个值都可以设为 `*`，分别表示启用来自所有主机或所有组的身份模拟。示例 12-2 展示了从主机 `hue.example.com`

以 hadoop-users 组的用户身份访问 Hadoop 时，如何使 Hue 进行用户模拟。

示例 12-2　在 core-site.xml 中为 Hadoop 配置 Hue 用户模拟

```
<property>
  <name>hadoop.proxyuser.hue.hosts</name>
  <value>hue.example.com</value>
</property>
<property>
  <name>hadoop.proxyuser.hue.groups</name>
  <value>hadoop-users</value>
</property>
```

HBase 和 Hive 使用 Hadoop 的模拟配置，但 Oozie 必须单独配置。如果想从 Hue 中使用 Oozie，必须在 oozie-site.xml 文件中设置 oozie.service.ProxyUserService.proxyuser.<*user*>.hosts 和 oozie.service.ProxyUserService.proxyuser.<*user*>.groups 属性。示例 12-3 展示了从主机 hue.example.com 以 hadoop-users 组的用户身份访问 Oozie 时，如何使 Hue 进行用户模拟。

示例 12-3　在 oozie-site.xml 中为 Oozie 配置 Hue 用户模拟

```
<property>
  <name>oozie.service.ProxyUserService.proxyuser.hue.hosts</name>
  <value>hue.example.com</value>
</property>
<property>
  <name>oozie.service.ProxyUserService.proxyuser.hue.groups</name>
  <value>hadoop-users</value>
</property>
```

如果使用 Hue 的搜索应用，还需要在 Solr 中启用用户模拟。具体方法是设置 /etc/default/solr 文件中的 SOLR_SECURITY_ALLOWED_PROXYUSERS、SOLR_SECURITY_PROXYUSER_<*user*>_HOSTS 和 SOLR_SECURITY_PROX YUSER_<*user*>_GROUPS 环境变量。示例 12-4 展示了从主机 hue.example.com 以 hadoop-users 组的用

户身份访问 Solr 的用户模拟配置启用。

示例 **12-4**　在 /etc/default/solr 中为 Solr 配置 Hue 用户模拟

```
SOLR_SECURITY_ALLOWED_PROXYUSERS=hue
SOLR_SECURITY_PROXYUSER_hue_HOSTS=hue.example.com
SOLR_SECURITY_PROXYUSER_hue_GROUPS=hadoop-users
```

12.1　Hue HTTPS

默认情况下，Hue 运行在明文 HTTP 上，这适用于概念证明或者客户端和 Hue 之间的网络完全可信的情况。然而对于大部分环境，强烈推荐配置 Hue 使用 HTTPS。这在客户端和 Hue 之间的网络不完全受信任时尤其重要，因为大部分 Hue 的认证后端支持通过浏览器表单输入用户名和密码。

幸运的是，在 Hue 中配置 HTTPS 很容易，只需配置 `ssl_certificate` 和 `ssl_private_key`，这两项都在 `hue.ini` 文件的 `desktop` 部分。二者都必须是 PEM 格式，而且私钥不能用口令加密，如示例 12-5 所示。

示例 **12-5**　配置 Hue 使用 HTTPS

```
[desktop]
ssl_certificate=/etc/hue/conf/hue.crt
ssl_private_key=/etc/hue/conf/hue.pem
```

Hue 目前不支持使用受口令保护的私钥。这意味着尽最大可能保护 Hue 的私钥是很重要的。确保密钥被 hue 用户所有并只能被其所有者读取（如 `chmod 400 /etc/hue/conf/hue.pm`）。也可以在文件系统中配置文件系统级加密，按照 9.2.6 节的方法存储私钥。如果 Hue 部署在拥有被 TLS/SSL 保护的其他资源的服务器上，建议发放一个仅供 Hue 使用的证书。

12.2　Hue身份验证

Hue 拥有可插拔的身份验证框架，其中包含很多身份验证后端。默认的身份验证后端使用支持数据库中存储的用户名密码私有列表。后端的配置需要将 [desktop] 节 [[auth]] 小节下的 backend 属性设置为 desktop.auth.backend.AllowFirstUserDjangoBackend。示例 12-6 明确设置了后端的 hue.ini 文件。这个设置是默认的，不必费心。

示例 **12-6** 配置默认的 Hue 身份验证后端

```
[desktop]
[[auth]]
backend=desktop.auth.backend.AllowFirstUserDjangoBackend
```

Hue 还支持使用 Kerberos/SPNEGO、LDAP、PAM 和 SAML 进行身份验证。此处不一一介绍，要了解更多信息请参考 config_help 命令。**1**

> 1根据 Hue 的安装方式，可以执行 /usr/share/hue/build/env/bin/hue config_help 或 /opt/cloudera/parcels/CDH/lib/hue/build/env/bin/hue config_help 以执行 config_help 命令。

12.2.1 SPNEGO后端

简单和受保护的 **GSSAPI** 协商机制（**SPNEGO**）**2** 是一个允许客户端和服务端协商选择身份验证技术的 GSSAPI 伪机制。客户端想在远程服务器上进行身份验证，但双方事先不知道对方支持的认证协议时，即可使用 SPNEGO。SPNEGO 最常用的地方是 Microsoft 最先提出的 HTTP 协商协议。**3**

> 2SPNEGO 伪机制的描述请参考 RFC 4178（http://tools.ietf.org/html/rfc4178）。

> 3详见 Microsoft 的 MSDN 文章（http://msdn.microsoft.com/en-us/library/ms995329.aspx）。

Hue 只支持使用 Kerberos V5 作为底层机制的 SPNEGO，这尤其意味着，Hue 无法与 **Microsoft NT LAN Manager**（**NTLM**）协议一起使用。为 SPNEGO 配置 Hue 时，需要将 Hue 身份验证后端设为 SpnegoDjangoBackend（示例 12-7），同时将 KRB5_KTNAME 环境变量设为含有 HTTP/< 完全限定域名 >@<REALM> 主体密钥的

keytab 文件位置。如果在服务器 hue.example.com 上手动启动 Hue，并且 keytab 文件位于 /etc/hue/conf/hue.keytab，那么应当如示例 12-8 所示启动 Hue。

示例 **12-7**　配置 SPNEGO Hue 身份验证后端

```
[desktop]
[[auth]]
backend=desktop.auth.backend.SpnegoDjangoBackend
```

示例 **12-8**　配置 KRB5_KTNAME 并人工启动 Hue

```
[hue@hue ~]$ export KRB5_KTNAME=/etc/hue/conf/hue.keytab
[hue@hue ~]$ ${HUE_HOME}/build/env/bin/supervisor
```

要使用 SPNEGO，还需要在桌面上拥有一个 TGT（例如通过运行 kinit 获得），并且需要使用支持 SPNEGO 的浏览器。Internet Explorer 和 Safari 都不需要额外配置就能支持 SPNEGO。如果使用 Firefox，首先必须将进行身份验证所用的服务器名或域名添加到受信 URI 列表。具体方法是，在 URL 栏键入 about:config，搜索 network.negotiate-auth.trusted-uris，然后更新该项使其包含该服务器名或域名。例如，如果想在 example.com 域的任意服务器中都支持 SPNEGO，那么需要设置 network.negotiate-auth.trusted-uris=example.com。如果尝试连接 Hue 时看到 401 Unauthorized 信息，那么可能没有在 Firefox 中正确配置受信 URI。

12.2.2　SAML后端

Hue 还支持使用安全断言置标语言（**Security Assertion Markup Language**，**SAML**）标准进行单点登录（**SSO**）。SAML 工作原理是，将服务提供者（**Service Provider**，**SP**）与身份提供者（**Identity Provider**，**IdP**）隔离。请求访问某个资源时，SP 会重定向到 IdP，并在那里进行身份验证；IdP 随后会向负责授予目标资源访问权限的 SP 传递一个验证身份的断言。维基百科关于 SAML 的文章（http://bit.ly/1GFpLur）里有更多细节，包括一个展示 SAML 过程步骤的图示。

配置 SAML 身份验证后端之后，Hue 就会作为服务提供者重定向到身份提供者，以进行身份验证。配置 Hue 使用 SAML 比使用其他身份

验证后端更为复杂。Hue 必须与一个第三方身份提供者进行交互，因此其中的一些细节就会取决于使用的是哪个身份提供者。另外， Hue 与一些使用 SAML 所需的依赖不兼容，因此先根据示例 12-9 中的步骤安装所需依赖。

示例 **12-9**　安装 SAML 身份验证后端的依赖

```
[root@hue ~]# yum install swig openssl openssl-devel gcc python-devel
Loaded plugins: fastestmirror, priorities
Loading mirror speeds from cached hostfile
 * base: mirror.hmc.edu
 * extras: mirrors.unifiedlayer.com
 * updates: mirror.pac-12.org
...
Complete!
[root@hue ~]# yum install xmlsec1 xmlsec1-openssl
Loaded plugins: fastestmirror, priorities
Loading mirror speeds from cached hostfile
 * base: mirror.hmc.edu
 * extras: mirrors.unifiedlayer.com
 * updates: mirror.pac-12.org
...
Complete!
[root@hue ~]# $HUE_HOME/build/env/bin/pip install --upgrade setuptools
Downloading/unpacking setuptools from https://pypi.python.org/packages/sou
e/s/setuptools/setuptools-7.0.tar.gz#md5=6245d6752e2ef803c365f560f7f2f940
  Downloading setuptools-7.0.tar.gz (793Kb): 793Kb downloaded
...
Successfully installed setuptools
Cleaning up...
[root@hue ~]# $HUE_HOME/build/env/bin/pip install -e \
  git+https://github.com/abec/pysaml2@HEAD#egg=pysaml2
Obtaining pysaml2 from git+https://github.com/abec/pysaml2@HEAD#egg=pysaml
  Updating ./build/env/src/pysaml2 clone (to HEAD)
...
Successfully installed pysaml2 m2crypto importlib WebOb
Cleaning up...
[root@hue ~]# $HUE_HOME/build/env/bin/pip install -e \
  git+https://github.com/abec/djangosaml2@HEAD#egg=djangosaml2
Obtaining djangosaml2 from git+https://github.com/abec/djangosaml2@HEAD#eg
djangosaml2
  Cloning https://github.com/abec/djangosaml2 (to HEAD) to ./build/env/src
jangosaml2
...
Successfully installed djangosaml2
Cleaning up...
```

如上安装一些开发工具，随后就会安装 SAML 工作所需的 Python 模块。安装完这些依赖之后，需要从身份提供者处下载元数据文件，具

体的细节取决于使用的是何种身份提供者。对于 Shibboleth 身份提供者，可以使用 `curl` 将 metadata 下载到 `/etc/hue/saml/metadata.xml`。

```
[root@hue ~]# mkdir /etc/hue/saml
[root@hue ~]# curl -k -o /etc/hue/saml/metadata.xml \
   https://idp.example.com:8443/idp/shibboleth
```

还需要一个证书和私钥对请求进行签名。这个密钥必须被身份提供者所信任，能够对请求进行签名，因此可能不能仅复用启用 Hue 的 HTTPS 时所用的密钥和证书。就本文而言，我们假定已经创建了密钥和证书，并且分别存放在 `/etc/hue/saml/key.pem` 和 `/etc/hue/saml/idp.pem` 文件中，剩下的就是配置 Hue 本身了。示例 12-10 给出了 `/etc/hue/conf/hue.ini` 文件中的相关部分。

示例 **12-10**　配置 SAML Hue 身份验证后端

```
[desktop]
[[auth]]
backend=libsaml.backend.SAML2Backend
[libsaml]
xmlsec_binary=/usr/bin/xmlsec1
create_users_on_login=true
metadata_file=/etc/hue/saml/metadata.xml
key_file=/etc/hue/saml/key.pem
cert_file=/etc/hue/saml/idp.pem
```

有一些额外可选的配置参数可以在 `saml` 配置组里设置，完整的配置参数列表如下。

`xmlsec_binary`

xmlsec1 程序的路径，该程序用于签名、验证、加密和解密 SAML 请求和断言。典型值为 `/usr/bin/xmlsec1`。

`create_users_on_login`

登录时创建 Hue 用户，可以是 `true` 或者 `false`。

`required_attributes`

Hue 必需的、从 IdP 获取的属性。其值是逗号分隔的属性列表。

例如：uid, email。

optional_attributes

Hue 可以处理的从 IdP 获取的属性。其值是逗号分隔的属性列表，处理方式与 required_attributes 一样。

metadata_file

从 IdP 获取的元数据 XML 文件的路径。该文件必须能够被 hue 用户读取。

key_file

PEM 格式的密钥文件。

cert_file

PEM 格式的 X.509 证书。

user_attribute_mapping

将从 IdP 接收到的属性（在 required_attributes、optional_attributes 和 IdP 配置中指定的）映射到 Hue 用户属性。例如：{*uid:*'username', email: *email*}。

authn_requests_signed

对身份验证请求进行签名。其值可以是 true 或 false，检查 IdP 文档，确认是否需要进行该项设置。

logout_requests_signed

对登出请求进行签名。其值可以是 true 或 false。检查 IdP 文档，确认是否需要进行该项设置。

12.2.3 LDAP后端

最后一种身份验证方式是使用 LDAP/Active Directory 验证用户名和口令。有两种配置 Hue 的 LDAP 后端的方法，第一种方法是进行 LDAP 搜索，找到用户的标识名（DN），然后使用 DN 绑定到 LDAP；第二种方法是向 Hue 提供一个填写了用户名的 DN 模式，之

后不用搜索即可绑定。

配置 Hue 使用搜索绑定时，必须将 search_bind_authentication 设为 true，并设置 desktop 节 ldap 小节中的 ldap_url 和 base_dn，还必须设置 desktop 节 ldap 小节下 users 小节中的 user_filter 和 user_name_attr，还应当设置 desktop 节 ldap 小节下 groups 小节中的 group_filter、group_name_attr 和 group_member_attr，这样才能将 LDAP 组导入为 Hue 组。

示例 12-11 展示了使用搜索绑定进行 LDAP 身份验证的 hue.ini 配置文件片段。示例中，LDAP 服务器运行在 ldap.example.com 的 LDAPS 上。该 LDAP 服务器将用户和组存储在一个基准 DN cn=accounts, dc=example,dc=com 之下。最后，用户账户在 objectClass=posixaccount 中，用户组在 objectClass=posixgroup 中。对于所有 LDAP 相关配置的完整描述见表 12-1。

示例 12-11　配置使用搜索绑定的 LDAP Hue 身份验证后端

```
[desktop]
[[auth]]
backend=desktop.auth.backend.LdapBackend

[[ldap]]
ldap_url=ldaps://ldap.example.com
base_dn="cn=accounts,dc=example,dc=com"
search_bind_authentication=true
ldap_cert=/etc/hue/conf/ca.crt
use_start_tls=false
create_users_on_login=true
[[[users]]]
user_filter="objectClass=posixaccount"
user_name_attr="uid"

[[[groups]]]
group_filter="objectClass=posixgroup"
group_name_attr="cn"
group_member_attr="member"
```

如果倾向于使用直接绑定，那么必须将 search_bind_authentication 设为 false，并且根据使用的是 Active Directory 还是其他 LDAP 提供者分别设置 nt_domain 或

者 ldap_username_pattern。仍然必须配置搜索相关的配置项（例如 user_filter、user_name_attr 等），因为从 LDAP 同步用户和组的时候会用到。如果想要使用与之前相同的服务器配置，但是用直接绑定而不是搜索绑定，那么与示例 12-12 类似。再次强调，LDAP 配置参数的完整列表见表 12-1。

示例 **12-12**　配置使用直接绑定的 LDAP Hue 身份验证后端

```
[desktop]
[[auth]]
backend=desktop.auth.backend.LdapBackend

[[ldap]]
ldap_url=ldaps://ldap.example.com
base_dn="cn=accounts,dc=example,dc=com"
search_bind_authentication=false
ldap_username_pattern="uid=<username>,cn=users,cn=accounts,dc=example,dc=c
ldap_cert=/etc/hue/conf/ca.crt
use_start_tls=false
create_users_on_login=true

[[[users]]]
user_filter="objectClass=posixaccount"
user_name_attr="uid"

[[[groups]]]
group_filter="objectClass=posixgroup"
group_name_attr="cn"
group_member_attr="member"
```

表12-1：Hue LDAP身份验证的配置属性

属性	描述
desktop.auth.backend	使用的身份验证后端（设为 desktop.auth.backend.LdapBackend）
desktop.ldap.ldap_url	LDAP 服务器 URL（对于安全 LDAP 使用 ldaps://）
desktop.ldap.base_dn	用于 LDAP 搜索的基本 LDAP 标识名
desktop.ldap.bind_dn	搜索 LDAP 时绑定的标识名；仅当 LDAP 服务器上的匿名搜索被禁用时需要
desktop.ldap.bind_password	bind_dn 用户的密码；仅当 LDAP 服务器上的匿名搜索被禁用时需要
desktop.ldap.create_users_on_login	设为 true，在首次登录时创建用户；如果设为 false，管理员必须手动向 Hue 添加用户，之后这些用户才能使用其 LDAP 凭证登录
desktop.ldap.search_bind_authentication	设为 true，使用搜索绑定；设为 false，启用直接绑定
desktop.ldap.ldap_username_pattern	从用户名创建标识名的模式——必须包含字符串 *<username>*，该值会被用户的用户名替代，从而创建最终的 DN；仅当配置 Hue 为直接绑定时（即 search_bind_authentication=false）使用
desktop.ldap.nt_domain	Active Directory 服务器的 NT 域；仅当配置 Hue 为直接绑定时（即 search_bind_authentication=*false*）使用
desktop.ldap.ldap_cert	验证 LDAP 服务器证书的 CA 证书位置

设为 true，使用 StartTLS；ldap_url 为 ldaps://URL 时，设为 false	
搜索用户时使用的基础过滤器	
LDAP 模式中的用户名属性（通常是 Active Diretory 的 sAMAccountName 和 LDAP 目录的 uid）	
搜索组时使用的基础过滤器	
LDAP 模式中的组属性（通常是 cn）	
指定组成员的 LDAP 属性（通常是 member）	

由示例 12-11 和示例 12-12 可知，将 ldap_cert 设置为指向一个 CA 证书，这是必需的，因为使用了 ldaps:// 的方式配置 LDAP URL。强烈建议使用 LDAPS 或 StartTLS。使用 StartTLS 时，要用 ldap:// 的方式配置 LDAP URL，并且将 use_start_tls 设为 true。

12.3 Hue授权

Hue 有两种用户账户：普通用户和超级用户。普通用户由基于访问控制表（ACL）的授权系统进行管理，该系统控制着哪些应用权限可用于哪些组。Hue 超级用户可以：

- 添加和删除用户
- 添加和删除组
- 给组分配权限
- 将某个用户变为超级用户
- 从 LDAP 服务器导入用户和组
- 安装查询、表和数据的样例
- 查看、提交和修改任意 Oozie 工作流、协调程序或 bundle
- 查看和修改 Sentry 权限时模拟任意用户
- 查看和修改 HDFS ACLs 时模拟任意用户

Hue 超级用户与 HDFS 超级用户不同。HDFS 超级用户是运行 NameNode 守护进程的用户，通常是 hdfs，并拥有列举、读取和写入任意 HDFS 文件和目录的权限。如果想从 Hue 上执行 HDFS 超级用户

操作，需要添加一个与 HDFS 超级用户同名的用户。或者可以设置 HDFS 超级组，以给一组用户分配 HDFS 超级用户权限。参考 6.1 节。

每个 Hue 应用定义了用户可以执行的一个或多个操作，授权控制通过为每个操作配置 ACL 实现，ACL 列出了可以执行该操作的组。每个应用都有一个名为“启动该应用”的操作，这决定了哪些用户可以运行该应用。一些应用还定义了额外的可控操作。

Hue 中授予的权限仅仅是从特定 Hue 应用调用特定操作的权限。执行某个操作的用户仍然需要由被他们访问的服务进行授权。例如，一个用户可能拥有 Metastore 应用中“允许 DDL 操作”的权限，但如果没有 Sentry 数据库中的 ALL 权限，依然无法创建表。

HBase 应用定义了“允许在 HBase 应用中写入”的操作，这授予了从 HBase 应用中添加行、添加单元、编辑单元、删除行和删除单元的权限。Metastore 应用定义了“允许 DDL 操作”的操作，这授予了从 metastore 浏览器创建、编辑和删除表的权限。Oozie 应用定义了“对所有作业的 Oozie 仪表盘只读用户”，这授予了对——无论是否被共享——所有工作流、协调器和 bundle 进行只读访问的权限。Security 应用定义了“列举文件或表等对象时，让一个用户模拟另外一个用户的身份”的操作，该操作允许用户模拟其他用户，并查看该用户能访问哪些表、文件和目录。

为“列举文件或表等对象时，让一个用户模拟另外一个用户的身份”授予权限，会暴露原本不可用的信息。特别地，它允许一个没有获得授权的已登录用户模拟另一个能够查看目录中文件的用户，授予执行该操作的权限时应当谨慎。另外值得提醒的是，Hue 超级用户还能够在 Security 应用中模拟用户，因此将一个用户变为 Hue 超级用户时也应当注意。

Useradmin 应用定义了“在 User Admin 上访问个人资料页面”的操作，但该操作已被弃用，可放心忽略。

12.4 Hue SSL客户端配置

之前介绍了很多安全相关的配置，但还没有详尽介绍如何在一般情况下建立和配置 Hue，这超出了本书范围。但一个重要问题是，各种底

层组件被配置成 SSL 在线加密时，如何恰当配置 Hue。如果 Hadoop、Hive 和 Impala 启用了 SSL，示例 12-13 展示了必要的配置片段。

示例 **12-13**　Hue SSL 配置

```
# Non-SSL configurations omitted for brevity
[beeswax]
 [[ssl]]
enabled=true
cacerts=/etc/hue/ca.cer
key=/etc/hue/host.key
cert=/etc/hue/host.cer
validate=true
[impala]
[[ssl]]
enabled=true
cacerts=/etc/hue/ca.cer
key=/etc/hue/host.key
cert=/etc/hue/host.cer
validate=true
```

Hive 和 Impala 配置中，`validate` 选项都会指定是否要求 Hue 验证哪些服务提供的证书是由配置的证书链中的机构签发的。

除示例之外，环境变量 `REQUESTS_CA_BUNDLE` 需要指向 SSL 证书链（PEM 格式）文件所在的磁盘位置。这被 Hadoop SSL 客户端用于访问 HDFS、MapReduce、YARN 和 HttpFS。

12.5　小结

本章详细讨论了允许终端用户通过集中式网络控制台访问各种生态系统组件时，Hue 的重要地位。在 Hue 的帮助下，用户只需登录网络控制台一次，剩下的集群操作都通过 Hue 系统用户的身份模拟执行。

本章对数据安全组件的讨论到此结束，接下来通过一些实际场景的案例学习，整合之前学到的知识。

第四部分　综合应用

第 13 章　案例分析

本章将介绍两个覆盖了书中很多安全主题的案例分析。首先，我们看一下如何在多重任务处理环境中使用 Sentry，以控制 SQL 访问数据。这在深入讨论一个更详细的案例前是个很好的热身，接下来的案例展示了一个自定义 HBase 应用以及各种安全特性。

13.1　案例分析：Hadoop 数据仓库

大数据和 Hadoop 的主要好处之一就是，可以汇集很多不同的数据集解决独特问题。随之而来的是跨越多个业务线的不同类型的用户。该案例分析中，我们将看看在包含多条业务线、多个数据所有者以及不同分析员的环境下，如何用 Sentry 提供强大的 Hive 和 Impala 数据授权。

首先列出这个案例分析的假设。

- 环境包含 3 条业务线，我们将其称为`lob1`、`lob2` 和 `lob3`。
- 每条业务线拥有分析员和管理员：
    - 分析员由组 `lob1grp`、`lob2grp` 和 `lob3grp` 定义；
    - 管理员由组 `lob1adm`、`lob2adm` 和 `lob3adm` 定义；
    - 管理员也在分析员组中。
- 每条业务线均需要在 HDFS 中拥有自己的沙箱区进行即席分析，并上传自服务的数据资源。
- 每条业务线拥有自己的管理员，控制对各自沙箱的访问权限。
- Hive 仓库中的数据是 IT 管理的，意味着只有非交互的 ETL 用户会添加数据。
- 只有 Hive 管理员会在 Hive 仓库中创建新的对象。
- Hive 仓库使用默认的 HDFS 路径 `/user/hive/warehouse`。

- 已经为集群设置 Kerberos。
- 环境中已经建立 Sentry。
- HDFS 已经启用扩展 ACL。
- HDFS 的默认权限掩码设置为 007。

13.1.1　环境搭建

有了基本假设后，需要在 HDFS 中为 Sentry 准备必要的目录。首先要做的就是锁定 Hive 仓库目录。启用 Sentry 时，HiveServer2 的身份模拟会被禁用，所以只有 `hive` 组具有访问权限（包括 `hive` 和 `impala` 用户）。我们需要做的如下所示：

```
[root@server1 ~]# kinit hive
Password for hive@EXAMPLE.COM:
[root@server1 ~]# hdfs dfs -chmod -R 0771 /user/hive/warehouse
[root@server1 ~]# hdfs dfs -chown -R hive:hive /user/hive/warehouse
[root@server1 ~]#
```

正如前面假设所言，每条业务线都需要一个沙箱区域。需要创建一个 `/data/sandbox` 作为所有沙箱的根目录，然后在其中创建相应的结构：

```
[root@server1 ~]# kinit hdfs
Password for hdfs@EC2.INTERNAL:
[root@server1 ~]# hdfs dfs -mkdir /data
[root@server1 ~]# hdfs dfs -mkdir /data/sandbox
[root@server1 ~]# hdfs dfs -mkdir /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -mkdir /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -mkdir /data/sandbox/lob3
[root@server1 ~]# hdfs dfs -chmod 770 /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -chmod 770 /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -chmod 770 /data/sandbox/lob3
[root@server1 ~]# hdfs dfs -chgrp lob1grp /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -chgrp lob2grp /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -chgrp lob3grp /data/sandbox/lob3
[root@server1 ~]#
```

基本的目录结构建好之后，需要开始思考，要用什么支持 `Hive` 和 `Impala` 访问沙箱，毕竟这些沙箱是所有用户要进行即席分析的地方。`hive` 和 `impala` 用户都需要访问这些目录，所以下面要创建

HDFS 扩展的 ACL（HDFS-extended ACL），以允许 `hive` 组的完全访问。

```
[root@server1 ~]# hdfs dfs -setfacl -m default:group:hive:rwx /data/sandbo
[root@server1 ~]# hdfs dfs -setfacl -m default:group:hive:rwx /data/sandbo
[root@server1 ~]# hdfs dfs -setfacl -m default:group:hive:rwx /data/sandbo
[root@server1 ~]# hdfs dfs -setfacl -m group:hive:rwx /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -setfacl -m group:hive:rwx /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -setfacl -m group:hive:rwx /data/sandbox/lob3
[root@server1 ~]#
```

记住，默认 ACL 只适用于目录，它仅规定了复制到新的子目录和文件中的 ACL。由于这个原因，父目录依然需要一个常规访问 ACL。

接下来需要确保，无论谁创建新文件，都能保持预期的访问权限不变。如果放任现在的权限不管，那么 `hive` 或 `impala` 用户新创建的目录和文件就可能被业务线中的分析员和管理员访问。为了解决这个问题，向扩展 ACL 添加以下组：

```
[root@server1 ~]# hdfs dfs -setfacl -m default:group:lob1grp:rwx \
  /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -setfacl -m default:group:lob1adm:rwx \
  /data/sandbox/lob1
[root@server1 ~]# hdfs dfs -setfacl -m default:group:lob2grp:rwx \
  /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -setfacl -m default:group:lob2adm:rwx \
  /data/sandbox/lob2
[root@server1 ~]# hdfs dfs -setfacl -m default:group:lob3grp:rwx \
  /data/sandbox/lob3
[root@server1 ~]# hdfs dfs -setfacl -m default:group:lob3adm:rwx \
  /data/sandbox/lob3
[root@server1 ~]# hdfs dfs -setfacl -m group:lob1grp:rwx /data/sandbox/lob
[root@server1 ~]# hdfs dfs -setfacl -m group:lob1adm:rwx /data/sandbox/lob
[root@server1 ~]# hdfs dfs -setfacl -m group:lob2grp:rwx /data/sandbox/lob
[root@server1 ~]# hdfs dfs -setfacl -m group:lob2adm:rwx /data/sandbox/lob
[root@server1 ~]# hdfs dfs -setfacl -m group:lob3grp:rwx /data/sandbox/lob
[root@server1 ~]# hdfs dfs -setfacl -m group:lob3adm:rwx /data/sandbox/lob
[root@server1 ~]#
```

所有扩展 ACL 已设置，查看其中之一：

```
[root@server1 ~]# hdfs dfs -getfacl -R /data/sandbox/lob1
# file: /data/sandbox/lob1
# owner: hdfs
# group: lob1grp
```

```
user::rwx
group::rwx
group:hive:rwx
group:lob1adm:rwx
group:lob1grp:rwx
mask::rwx
other::---
default:user::rwx
default:group::rwx
default:group:hive:rwx
default:group:lob1adm:rwx
default:group:lob1grp:rwx
default:mask::rwx
default:other::---
[root@server1 ~]#
```

我们已经处理了所有租用集群的用户，还要确认也在 HDFS 中为非交互的 ETL 用户创建了可用空间：

```
[root@server1 ~]# hdfs dfs -mkdir /data/etl
[root@server1 ~]# hdfs dfs -chown etluser:hive /data/etl
[root@server1 ~]# hdfs dfs -chmod 770 /data/etl
[root@server1 ~]# hdfs dfs -setfacl -m default:group:hive:rwx /data/etl
[root@server1 ~]# hdfs dfs -setfacl -m group:hive:rwx /data/etl
[root@server1 ~]# hdfs dfs -setfacl -m default:user:etluser:rwx /data/etl
[root@server1 ~]# hdfs dfs -setfacl -m user:etluser:rwx /data/etl
[root@server1 ~]# hdfs dfs -getfacl /data/etl
# file: /data/etl
# owner: etluser
# group: hive
user::rwx
user:etluser:rwx
group::rwx
group:hive:rwx
mask::rwx
other::---
default:user::rwx
default:user:etluser:rwx
default:group::rwx
default:group:hive:rwx
default:mask::rwx
default:other::---
[root@server1 ~]#
```

下一步是开始在 Hive 中使用 `beeline` 命令行执行一些管理任务。我们将使用 `hive` 用户，因为默认情况下它是一个 Sentry 管理员，因此可以创建策略。

你可以使用属性文件为 `beeline` 指定连接信息。这比记住语法或者查找 bash 历史记录容易得多。

我们要使用的 `beeline.properties` 文件如示例 13-1 所示。注意，用户名和口令是实际认证所必需的，但空着，因为启用了 Kerberos。

示例 **13-1**　beeline.properties 文件

```
ConnectionURL=jdbc:hive2://server1.example.com:10000/;principal=
  hive/server1.example.com@EXAMPLE.COM
ConnectionDriverName=org.apache.hive.jdbc.HiveDriver
ConnectionUserName=.
ConnectionPassword=.

    [root@server1 ~]# kinit hive
    Password for hive@EXAMPLE.COM:
    [root@server1 ~]# beeline
    ...
    beeline> !properties beeline.properties
    ...
    > CREATE ROLE sqladmin;
    > GRANT ROLE sqladmin TO GROUP hive;
    > GRANT ALL ON SERVER server1 TO ROLE sqladmin;
    > CREATE DATABASE lob1 LOCATION '/data/sandbox/lob1';
    > CREATE DATABASE lob2 LOCATION '/data/sandbox/lob2';
    > CREATE DATABASE lob3 LOCATION '/data/sandbox/lob3';
    > CREATE DATABASE etl LOCATION '/data/etl';
```

创建管理员角色和数据库之后，可以设置 Sentry 策略为 Hive 和 Implala 提供对终端用户的授权。

```
    > CREATE ROLE lob1analyst;
    > GRANT ROLE lob1analyst TO GROUP lob1grp;
    > GRANT ALL ON DATABASE lob1 TO ROLE lob1analyst;
    > CREATE ROLE lob1administrator;
    > GRANT ROLE lob1administrator TO GROUP lob1adm WITH GRANT OPTION;
    > GRANT ALL ON DATABASE lob1 TO role lob1administrator;
    > CREATE ROLE lob2analyst;
    > GRANT ROLE lob2analyst TO GROUP lob2grp;
    > GRANT ALL ON DATABASE lob2 TO ROLE lob2analyst;
    > CREATE ROLE lob2administrator;
    > GRANT ROLE lob2administrator TO GROUP lob2adm WITH GRANT OPTION;
    > GRANT ALL ON DATABASE lob2 TO ROLE lob2administrator;
    > CREATE ROLE lob3analyst;
    > GRANT ROLE lob3analyst TO GROUP lob3grp;
    > GRANT ALL ON DATABASE lob3 TO role lob3analyst;
```

```
> CREATE ROLE lob3administrator;
> GRANT ROLE lob3administrator TO GROUP lob3adm WITH GRANT OPTION;
> GRANT ALL ON DATABASE lob3 TO ROLE lob3administrator;
> CREATE ROLE etl;
> GRANT ROLE etl TO GROUP etluser;
> GRANT ALL ON DATABASE etl TO ROLE etl;
```

列在假设里的另一个重要需求是，用户应能上传自服务文件到他们自己的沙箱。要允许用户利用 Hive 和 Impala 中的这些文件，他们还需要一些 URI 权限。我们还将继续提供写权限，使用户能够从 Hive 提取数据并导入沙箱区域，以供额外的非 SQL 分析使用。

```
> GRANT ALL ON URI 'hdfs://nameservice1/data/etl' TO ROLE etl;
> GRANT ALL ON URI 'hdfs://nameservice1/data/sandbox/lob1' TO ROLE lob
> GRANT ALL ON URI 'hdfs://nameservice1/data/sandbox/lob1' TO ROLE lob
> GRANT ALL ON URI 'hdfs://nameservice1/data/sandbox/lob2' TO ROLE lob
> GRANT ALL ON URI 'hdfs://nameservice1/data/sandbox/lob2' TO ROLE lob
> GRANT ALL ON URI 'hdfs://nameservice1/data/sandbox/lob3' TO ROLE lob
> GRANT ALL ON URI 'hdfs://nameservice1/data/sandbox/lob3' TO ROLE lob
```

以上展示的 URI 路径使用了 HDFS HA 命名服务名称。如果没有配置 HA，则需要指定 NameNode 的完全限定域名，包括端口（8020）。

13.1.2 用户体验

当环境全部上线就绪，并且配备一整套 HDFS 权限和 Sentry 策略后，下面看看在这些落实的措施下，终端用户看到的内容。首先看看 `sqladmin` 角色的用户会看到什么：

```
[root@server1 ~]$ kinit hive
Password for hive@EXAMPLE.COM:
[root@server1 ~]$ beeline
...
> !properties beeline.properties
...
> SHOW DATABASES;
+-----------------+
| database_name   |
+-----------------+
| default         |
| etl             |
| lob1            |
| lob2            |
| lob3            |
```

```
+----------------+
> quit;
[root@server1 ~]$
```

如上所示，sqladmin 角色被允许查看我们创建的每一个数据库。这和预期的一致，因为 sqladmin 角色被授予了对 SERVER 对象的完全访问权限。接下来看看被分配到 etl 角色的用户会看到什么：

```
[root@server1 ~]$ kinit etluser
Password for etluser@EXAMPLE.COM:
[root@server1 ~]$ beeline
...
> !properties beeline.properties
...
> SHOW DATABASES;
+----------------+
| database_name  |
+----------------+
| default        |
| etl            |
+----------------+
> USE lob1;
Error: Error while compiling statement: FAILED: SemanticException
 No valid privileges (state=42000,code=40000)
> quit;
[root@server1 ~]$
```

用户这次没有看到 metastore 中的所有数据库，而只看到了包含他们具有访问权限的对象的数据库。上述例子说明，不仅用户没有访问权限的对象是对用户隐藏的，而且即便用户通过名称请求该对象，也会被拒绝。这正是我们期望的。

现在，假定 etl 数据库中的表 sample_07 需要变成对 lob1analyst 角色可用，然而要注意：并非所有列都可以共享。为此，需要创建一个只包含我们想对该角色可见的列的视图。创建该视图后，授予 lob1analyst 角色对其的访问权限：

```
[root@server1 ~]$ kinit hive
Password for hive@EXAMPLE.COM:
[root@server1 ~]$ beeline
...
> !properties beeline.properties
...
> USE etl;
> CREATE VIEW sample_07_view AS SELECT code, description, total_emp
FROM sample_07;
```

```
> GRANT SELECT ON TABLE sample_07_view TO ROLE lob1analyst;
> quit;
[root@server1 ~]$
```

完成这些工作后，可以用一个被分配到 lob1analyst 角色的用户测试访问权限：

```
[root@server1 ~]$ kinit lob1user
Password for lob1user@EXAMPLE.COM:
[root@server1 ~]$ beeline
...
> !properties beeline.properties
...
> SHOW DATABASES;
+-----------------+
| database_name   |
+-----------------+
| default         |
| etl             |
| lob1            |
+-----------------+
> USE etl;
> SHOW TABLES;
+-----------------+
|    tab_name     |
+-----------------+
| sample_07_view  |
+-----------------+
> SELECT * FROM sample_07 LIMIT 1;
Error: Error while compiling statement: FAILED: SemanticException
 No valid privileges (state=42000,code=40000)
> quit;
[root@server1 ~]$ hdfs dfs -ls /data/etl
ls: Permission denied: user=lob1user, access=READ_EXECUTE, inode="/data/et
 etluser:hive:drwxrwx---:group::---,group:hive:rwx,
 default:user::rwx,default:group::---,default:group:hive:rwx,
 default:mask::rwx,default:other::---
[root@server1 ~]$
```

如上所示，lob1user 能够在列表中看到 etl 数据库。但注意，该数据库中只有 sample_07_view 对象是可见的。如我们所预期的，用户无法通过 SQL 访问或者直接的 HDFS 访问读取源表。上述例子中有一些 access denied 信息，所以查看日志文件中出现了什么。从 HiveServer2 日志开始：

```
2015-01-13 19:31:40,173 ERROR org.apache.hadoop.hive.ql.Driver: FAILED:
 SemanticException No valid privileges
```

```
org.apache.hadoop.hive.ql.parse.SemanticException: No valid privileges
        at org.apache.sentry.binding.hive.HiveAuthzBindingHook.
         postAnalyze(HiveAuthzBindingHook.java:320)
        at org.apache.hadoop.hive.ql.Driver.compile(Driver.java:457)
        at org.apache.hadoop.hive.ql.Driver.compile(Driver.java:352)
        at org.apache.hadoop.hive.ql.Driver.compileInternal
         (Driver.java:995)
        at org.apache.hadoop.hive.ql.Driver.compileAndRespond
         (Driver.java:988)
        at org.apache.hive.service.cli.operation.SQLOperation.prepare
         (SQLOperation.java:98)
        at org.apache.hive.service.cli.operation.SQLOperation.run
         (SQLOperation.java:163)
        at org.apache.hive.service.cli.session.HiveSessionImpl.
         runOperationWithLogCapture(HiveSessionImpl.java:524)
        at org.apache.hive.service.cli.session.HiveSessionImpl.
         executeStatementInternal(HiveSessionImpl.java:222)
        at org.apache.hive.service.cli.session.HiveSessionImpl.
         executeStatement(HiveSessionImpl.java:204)
        at org.apache.hive.service.cli.CLIService.executeStatement
         (CLIService.java:168)
        at org.apache.hive.service.cli.thrift.ThriftCLIService.
         ExecuteStatement(ThriftCLIService.java:316)
        at org.apache.hive.service.cli.thrift.TCLIService$Processor
         $ExecuteStatement.getResult(TCLIService.java:1373)
        at org.apache.hive.service.cli.thrift.TCLIService$Processor
         $ExecuteStatement.getResult(TCLIService.java:1358)
        at org.apache.thrift.ProcessFunction.process
         (ProcessFunction.java:39)
        at org.apache.thrift.TBaseProcessor.process(TBaseProcessor.java:39
        at org.apache.hadoop.hive.thrift.HadoopThriftAuthBridge20S$Server
         $TUGIAssumingProcessor.process(HadoopThriftAuthBridge20S.java:608
        at org.apache.thrift.server.TThreadPoolServer$WorkerProcess.run
         (TThreadPoolServer.java:244)
        at java.util.concurrent.ThreadPoolExecutor.runWorker
         (ThreadPoolExecutor.java:1145)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run
         (ThreadPoolExecutor.java:615)
        at java.lang.Thread.run(Thread.java:745)
Caused by: org.apache.hadoop.hive.ql.metadata.AuthorizationException:
         User lob1user does not have privileges for QUERY
        at org.apache.sentry.binding.hive.authz.HiveAuthzBinding.authorize
         (HiveAuthzBinding.java:317)
        at org.apache.sentry.binding.hive.HiveAuthzBindingHook.
         authorizeWithHiveBindings(HiveAuthzBindingHook.java:502)
        at org.apache.sentry.binding.hive.HiveAuthzBindingHook.
         postAnalyze(HiveAuthzBindingHook.java:312)
        ... 20 more
```

接下来，可以看到出现在 NameNode 审计日志中的拒绝访问的审计事件：

```
2015-01-13 20:01:15,005 INFO FSNamesystem.audit: allowed=false
  ugi=lob1user@EXAMPLE.COM (auth:KERBEROS)
ip=/10.6.9.73
 cmd=listStatus src=/data/etl dst=null perm=null
```

13.1.3　小结

这个基础的案例分析展示了，如何看待使用 Sentry 策略外加 HDFS 扩展 ACL 进行数据保护。该示例有意做得很简单，但它也说明，在多重任务处理环境下，将数据组织作为一个关键因素考虑何等重要。只有对数据如何存放在 HDFS 中具有一个清晰的概念，安全管理才会更容易。

13.2　案例分析：交互式HBase Web应用

Hadoop 的一个常见用例是搭建大规模 Web 应用。HBase 的一些特性使它成为交互式大规模应用的理想选择：

- 支持复杂对象、具有快速演化模式的灵活的数据模型
- 添加或移除集群节点时的数据自动重分区
- 集成 Hadoop 生态系统其余部分以支持事务数据的离线分析
- 行内 ACID 交易
- 对各种应用的高级授权功能

对于本书，我们最关心列表中的最后一个功能。对于交互式应用，你经常需要对哪个用户能访问哪个数据集进行控制。例如，一个类似 Twitter 的应用就具有完全公开的消息、仅限授权用户白名单的消息，以及完全私有的消息。要想在这种动态安全需求面前灵活管理授权，需要同样动态的数据库使用方式。

这个案例分析中，我们将看到一个存储和浏览网页快照的应用。这个案例分析建立在 The Kite SDK（http://kitesdk.org）提供的一个基于 HBase 的开源 Web 应用示例（https://github.com/kite-sdk/kite-spring-hbase-example），原本的示例作为部署在 OpenShift 中的一个应用，以及部署在 HBase 集群中的一个生产应用，工作在单机开发模式下。由于 `MiniHBaseCluster` 类（用于开发模式和 OpenShift

部署）的局限，我们的版本仅为生产应用，保护 HBase 集群。我们这个版本的该示例完整源代码在与本书配套的 GitHub 源码库（https://github.com/hadoop-security/kite-spring-hbase-example）中。

13.2.1 设计与架构

首先看看这个网页快照示例的架构，如图 13-1 所示。该 Web 应用部署在一个边界节点上，用户通过浏览器连接到该应用，并使用 URL 创建新快照或者查看已有快照。创建新快照时，该 Web 应用会下载网页和元数据，并将它们存储在 HBase 中。查看快照时，该 Web 应用会从 HBase 获取页面元数据和快照，并展示到浏览器。

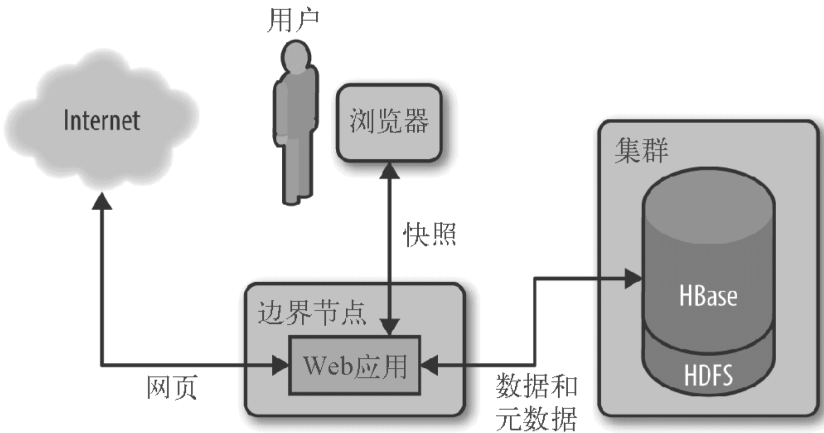


图 **13-1**：**Web** 应用架构

开始讨论安全需求之前，先看看该例中使用的数据模型。每个网页都使用一个 URL 作为唯一标识，每个快照由获取网页的时间进一步确定。数据模型中的完整字段列表见表 13-1。

表**13-1**：网页快照数据模型

url	String	网页的 URL
timestamp	long	网页被获取的 UTC 时间
fetchTimeMs	int	获取网页所用的时间，以 ms 为单位
size	int	网页大小
title	String	页面标题（如果有）
description	String	元标签（meta tag）中的描述
keywords	List<String>	元标签中的关键字
outlinks	List<String>	该网页链接到的页面URL
content	String	网页内容

HBase 将数据存储为多维排序映射，这意味着需要将我们记录里的字段映射到 Habse 用于排列数据的行键、列族和列限定键。对于用例而言，我们希望 HBase 的每行以 URL 和获取快照的时间为键值进行检索。为了使最近的快照排在前面，我们在行键中使用 `fetchedAt` 时间戳之前，将先使用 `Long.MAX_VALUE` 减去其值，从而实现反向排序，见图 13-2 中的 `<rev fetchedAt>`。每个字段都对应 HBase 中的一列，因此定义了从每个字段名称到列族和列限定符的映射。图 13-2 展示了行键是如何被映射的，并给出了映射到 HBase 列的字段示例。

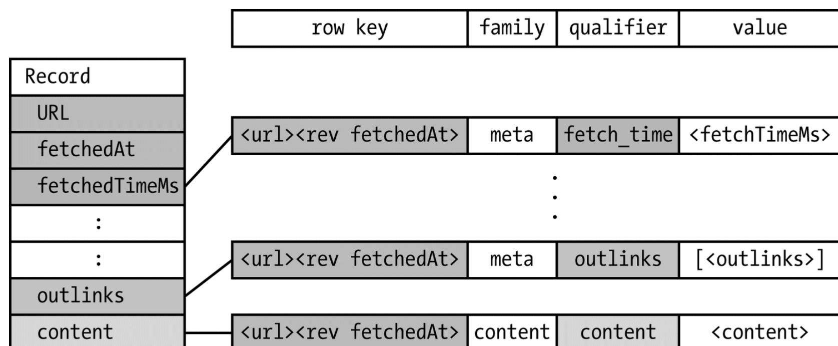


图 **13-2**：原始的 **HBase** 数据模型映射

13.2.2 安全需求

现在可以向该示例添加安全特性了。默认情况下，快照中的所有字段都能被任意用户访问。我们的用例中，想在默认情况下锁定网页内容，只在我们要求公开某个快照时才能被访问。可以使用单元级安全并保持数据模型与之前使用的一样，但这对于用例而言可能是“杀鸡用牛刀”。相反，我们可以稍微修改一下数据模型。

特别地，我们将向模型添加一个 `contentKey` 字段，`contentKey` 将被用作存储内容的列限定符。使用用户名作为私有快照的 `contentKey`，对于公开快照则使用特殊值 `public`。现在希望在可能的不同列限定符中存储每个快照的内容。因此，将 `content` 字段的类型改为 `Map<String, String>`。图 13-3 展示了更新后的映射配置。

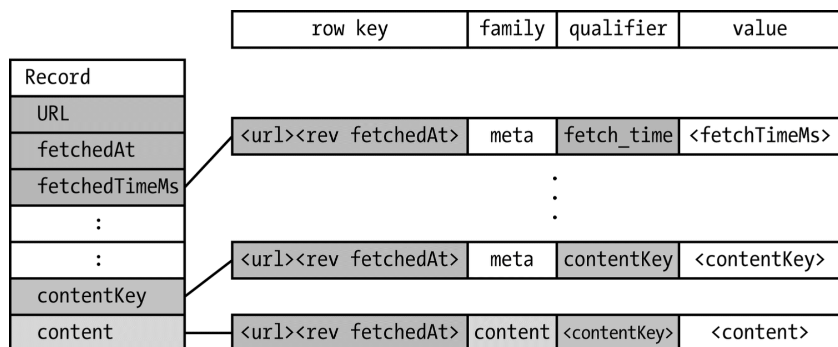


图 **13-3**：更新后的 **HBase** 数据模型映射

继续之前，先总结一下想在应用中实施的安全需求列表：

(1) 私有快照的内容只能被创建该快照的用户访问

(2) 公开快照的内容对所有用户可见

(3) 所有快照的元数据对所有用户可见

(4) 用户使用 HTTP 基本认证进行应用的身份验证

(5) 应用模拟已认证用户的身份与 HBase 通信

(6) 在 HBase 级进行强制授权

13.2.3　集群配置

有了这些需求，下面可以开始配置集群。需求 (5) 说明该应用需要与 HBase 进行身份验证。为了启用 HBase 身份验证，必须先启用 Hadoop 身份验证。要满足需求 (6)，还需要启用 HBase 授权，HBase 授权也是需求 (1) 和 (2) 的必要条件。需求 (3) 说明，要允许所有用户访问元数据字段。需求 (4) 适用于 Web 应用自身以及应用服务器，对于我们而言，使用的是 Tomcat。现在准备开始计划我们的配置步骤：

(1) 配置 Hadoop 身份验证（见 5.2.5 节）；

(2) 配置 HBase 身份验证［参考 *The Apache HBase Reference Guide*（http://hbase.apache.org/book.html）中的 Securing Apache HBase（http://hbase.apache.org/book.html#security）］；

(3) 向 `hbase-site.xml` 添加如下内容，以配置 HBase 授权：

```
<property>
  <name>hbase.coprocessor.region.classes</name>
  <value>
    org.apache.hadoop.hbase.security.access.AccessController,
    org.apache.hadoop.hbase.security.token.TokenProvider
  </value>
</property>
<property>
  <name>hbase.coprocessor.master.classes</name>
  <value>
    org.apache.hadoop.hbase.security.access.AccessController
  </value>
</property>
<property>
  <name>hbase.coprocessor.regionserver.classes</name>
  <value>
    org.apache.hadoop.hbase.security.access.AccessController
  </value>
</property>
<property>
  <name>hbase.security.exec.permission.checks</name>
  <value>true</value>
</property>
```

(4) 创建一个 Kerberos 主体，执行 HBase 管理功能：

```
kadmin: addprinc hbase@EXAMPLE.COM
WARNING: no policy specified for hbase@EXAMPLE.COM; defaulting to no \
policy
Enter password for principal "hbase@EXAMPLE.COM":
Re-enter password for principal "hbase@EXAMPLE.COM":
Principal "hbase@EXAMPLE.COM" created.
kadmin:
```

(5) 为该应用创建一个 Kerberos 主体，并将密钥导出到 keytab 文件：

```
kadmin: addprinc web-page-snapshots@EXAMPLE.COM
WARNING: no policy specified for web-page-snapshots@EXAMPLE.COM;
defaulting to no policy
Enter password for principal "web-page-snapshots@EXAMPLE.COM":
Re-enter password for principal "web-page-snapshots@EXAMPLE.COM":
Principal "web-page-snapshots@EXAMPLE.COM" created.
kadmin: ktadd -k app.keytab web-page-snapshots
Entry for principal web-page-snapshots with kvno 4, encryption type
des3-cbc-sha1 added to keytab WRFILE:app.keytab.
```

```
Entry for principal web-page-snapshots with kvno 4, encryption type
arcfour-hmac added to keytab WRFILE:app.keytab.
Entry for principal web-page-snapshots with kvno 4, encryption type
des-hmac-sha1 added to keytab WRFILE:app.keytab.
Entry for principal web-page-snapshots with kvno 4, encryption type
des-cbc-md5 added to keytab WRFILE:app.keytab.
kadmin:
```

(6) 将 `keytab` 文件复制到该应用用户的 home 目录；

(7) 授予应用主体创建表的权限：

```
[app@snapshots ~]$ kinit hbase
Password for hbase@ENT.CLOUDERA.COM:
[app@snapshots ~]$ hbase shell
14/11/13 14:45:53 INFO Configuration.deprecation: hadoop.native.lib is
deprecated. Instead, use io.native.lib.available
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.6, rUnknown, Sat Oct 11 15:15:15 PDT 2014

hbase(main):001:0> grant 'web-page-snapshots', 'RWXCA'
0 row(s) in 4.0340 seconds

hbase(main):002:0>
```

(8) 创建 HBase 表：

```
[app@snapshots ~]$ kinit -kt ~/app.keytab web-page-snapshots
[app@snapshots ~]$ export KITE_USER_CLASSPATH=/etc/hadoop/conf
[app@snapshots ~]$ export \
ZK=zk1.example.com,zk2.example.com,zk3.example.com
[app@snapshots ~]$ kite-dataset create \
  dataset:hbase:${ZK}:2181/webpagesnapshots.WebPageSnapshotModel \
  -s src/main/avro/hbase-models/WebPageSnapshotModel.avsc
[app@snapshots ~]$ kite-dataset create \
  dataset:hbase:${ZK}:2181/webpageredirects.WebPageRedirectModel \
  -s src/main/avro/hbase-models/WebPageRedirectModel.avsc
  [app@snapshots ~]$
```

(9) 授予用户 `alice` 和 `bob` 访问公共表 / 列的权限：

```
[app@snapshots ~]$ kinit -kt ~/app.keytab web-page-snapshots
[app@snapshots ~]$ hbase shell
14/11/13 14:45:53 INFO Configuration.deprecation: hadoop.native.lib is
deprecated. Instead, use io.native.lib.available
```

```
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.6, rUnknown, Sat Oct 11 15:15:15 PDT 2014

hbase(main):001:0> grant 'alice', 'RW', 'webpagesnapshots', 'content',
'public'
0 row(s) in 2.9580 seconds

hbase(main):002:0> grant 'alice', 'RW', 'webpagesnapshots', '_s'
0 row(s) in 0.1640 seconds

hbase(main):003:0> grant 'alice', 'RW', 'webpagesnapshots', 'meta'
0 row(s) in 0.2100 seconds

hbase(main):004:0> grant 'alice', 'RW', 'webpagesnapshots', 'observable'
0 row(s) in 0.1600 seconds

hbase(main):005:0> grant 'alice', 'RW', 'webpageredirects'
0 row(s) in 0.1600 seconds

hbase(main):006:0> grant 'alice', 'RW', 'managed_schemas'
0 row(s) in 0.1570 seconds

hbase(main):007:0> grant 'bob', 'RW', 'webpagesnapshots', 'content',
'public'
0 row(s) in 0.1920 seconds

hbase(main):008:0> grant 'bob', 'RW', 'webpagesnapshots', '_s'
0 row(s) in 0.1510 seconds

hbase(main):009:0> grant 'bob', 'RW', 'webpagesnapshots', 'meta'
0 row(s) in 0.2100 seconds

hbase(main):010:0> grant 'bob', 'RW', 'webpagesnapshots', 'observable'
0 row(s) in 0.1640 seconds

hbase(main):011:0> grant 'bob', 'RW', 'webpageredirects'
0 row(s) in 0.1590 seconds

hbase(main):012:0> grant 'bob', 'RW', 'managed_schemas'
0 row(s) in 0.1870 seconds

hbase(main):013:0>
```

(10) 授予 alice 和 bob 访问各自私有列的权限：

```
[app@snapshots ~]$ kinit -kt ~/app.keytab web-page-snapshots
[app@snapshots ~]$ hbase shell
14/11/13 14:45:53 INFO Configuration.deprecation: hadoop.native.lib is
deprecated. Instead, use io.native.lib.available
HBase Shell; enter 'help<RETURN>' for list of supported commands.
```

```
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.6, rUnknown, Sat Oct 11 15:15:15 PDT 2014

hbase(main):001:0> grant 'alice', 'RW', 'webpagesnapshots', 'content',
'alice'

0 row(s) in 2.8890 seconds
hbase(main):002:0> grant 'bob', 'RW', 'webpagesnapshots', 'content',
'bob'
0 row(s) in 0.1600 seconds

hbase(main):003:0>
```

(11) 添加如下参数到所有 HBase 节点的 `hbase-site.xml` 文件，从而启用利用 `web-page-snapshots` 主体的用户模拟：

```
<property>
  <name>hadoop.proxyuser.web-page-snapshots.groups</name>
  <value>*</value>
</property>
<property>
  <name>hadoop.proxyuser.web-page-snapshots.hosts</name>
  <value>*</value>
</property>
```

还有一些本示例应用的设计和实现所特有的其他配置步骤，运行该示例的完整步骤可参见 Github 上本项目的 README 文档（https://github.com/hadoop-security/kite-spring-habse-example）。

13.2.4　实现中的注意事项

为应用添加安全特性时，我们做了一些实现上的修改。虽然可以通过比较我们的示例与原始的 Kite SDK 示例查看完整的修改列表，但此处依然总结一下关键的修改。第一处修改是增加了额外的 Kerberos 登录模块，以通过应用的 keytab 获取 Kerberos TGT，Spring 初始化其他 Web 应用之前会加载该模块。下面是该模块的一个不包括登录或错误检查的简化版：

```
public class KerberosLoginService {

  public KerberosLoginService(String applicationPrincipal,
      String applicationKeytab) throws IOException {

    if (UserGroupInformation.isSecurityEnabled()) {
```

```
      UserGroupInformation.loginUserFromKeytab(applicationPrincipal,
          applicationKeytab);
    }
  }
}
```

这里的要点是，我们使用 `UserGroupInformation` 类的 `loginUserFromKeytab()` 方法前，要首先确认已经启用了集群上的安全策略。该方法会使用 keytab 文件获取我们的 Kerberos TGT。

从 Hadoop 安全角度考虑的第二处所需的改动是，修改 `WebPageSnapshotService` 以模拟认证后的用户与 HBase 通信。为了实现这个目标，使用 `UserGroupInformation` 对象的 `doAs()` 方法，该对象代表了我们想要模拟的代理用户。如下示例向 `WebPageSnapshotService` 中的一个方法添加身份模拟：

```
private WebPageSnapshotModel getWebPageSnapshot(String url,
    final long ts, final String user) throws IOException {
  WebPageSnapshotModel snapshot = null;
  final String normalizedUrl = normalizeUrl(url, user);

  UserGroupInformation ugi = UserGroupInformation.createProxyUser(user,
      UserGroupInformation.getLoginUser());
  snapshot = ugi.doAs(new PrivilegedAction<WebPageSnapshotModel>() {

    @Override
    public WebPageSnapshotModel run() {
      Key key = new Key.Builder(webPageSnapshotModels(user))
          .add("url", normalizedUrl)
          .add("fetchedAtRevTs", Long.MAX_VALUE - ts).build();
      return webPageSnapshotModels(user).get(key);
    }
  });

  return snapshot;
}
```

最后一个所需的改动是，从使用单一共享的 HBase 连接变成为每个用户创建一个连接，这是由 HBase 客户端缓存连接的方法决定的。最重要的关键点是，为每个用户创建连接，并在 HBase 要使用的 `Configuration` 对象中，将 `hbase.client.instance.id` 设为唯一值。对于该应用，我们创建一个实用方法以创建和缓存连接：

```
private synchronized RandomAccessDataset<WebPageSnapshotModel>
  webPageSnapshotModels(String user) {
```

```
  RandomAccessDataset<WebPageSnapshotModel> dataset =
      webPageSnapshotModelMap.get(user);

  if (dataset == null) {
    Configuration conf = new Configuration(
        DefaultConfiguration.get());
    conf.set("hbase.client.instance.id", user);
    DefaultConfiguration.set(conf);
    dataset = Datasets.load(webPageSnapshotUri,
        WebPageSnapshotModel.class);
    webPageSnapshotModelMap.put(user, dataset);
  }

  return dataset;
}
```

13.2.5　小结

本案例分析中，我们对一个典型交互式 HBase 应用的设计和架构进行了综述，然后探讨了与用例相关的安全考虑（身份验证、授权、身份模拟等），还描述了要支持我们的授权模型所必需的数据模型改动。接下来，总结了想要增加到应用中的安全需求，并给出配置集群，以满足该安全需求的必要步骤。最后，描述了应用实现中要支持这些安全需求所需的部分修改。

后记

问世以来，Hadoop 已经走过很长一段路。正如你在本书中看到的，安全在整个生态系统中包含大量内容。随着大数据的兴起，以及在那些迅速使用 Hadoop 作为数据平台的企业中的影响，Hadoop 及其庞大生态系统的快速发展已不足为奇。尽管如此，Hadoop 仍处于非常早期的阶段。虽然有很多可用的安全配置，但 Hadoop 要想达到关系型数据库和数据仓库的水准，以完全满足资产上亿的公司在数据管理上的需求，还有很多工作要做。

好在由于 Hadoop 在市场上的巨大增长，人们在快速填补其安全空缺。书中没有讨论的要么是现在正处于开发阶段（可能在本书出版时已经完成）的内容，要么是在不久的将来会变成 Hadoop 生态系统一部分的特性。

统一授权

Hadoop 安全管理员最困难的工作之一就是，要跟踪无数组件是如何处理访问控制的。虽然我们把很多东西加到 Apache Sentry，以作为 Hadoop 的集中授权组件，但在为整个生态系统提供授权方面还不够。从长期来看这是迟早的，也是必需的。安全管理员和审计员都需要有一个地方供他们查看和管理所有用户授权控制相关的策略，如果缺少这个，就很容易在过程中犯错。

短期而言，Apache Sentry 将会集成 HDFS 授权，这将允许有一个统一的方法定义在组件之间共享数据时的数据访问策略。例如，如果数据被载入 Hive 仓库并由 Sentry 策略控制，那将如何处理 MapReduce 访问？正如第 13 章所述，这涉及使用扩展 HDFS ACL。有了与 Sentry 集成的 HDFS 后，HDFS 路径可以被指定为由 Sentry 控制。因此，授权决策取决于 Sentry 策略，而不是标准 POSIX 权限或者扩展 ACL。

另外，Sentry 即将与 HBase 集成。我们在第 6 章看到，授权策略存储在 HBase 的一个特殊表中，并在默认情况下通过 HBase 命令行进行管理。将策略存储迁移到 Sentry 是一个好的选择。

数据管控

本书不讨论数据管控的大型话题，但讨论了其中一个关于审计的子话题。如第 8 章所述，集群中很多捕捉事件的地方都有审计日志，但没有集中的地方全面捕捉审计，也没有地方进行常规的数据管控任务，如管理业务元数据、查看联系和族谱、管理数据留存等。传统数据仓库中都突出包含了这些功能，为了使 Hadoop 能够在整体安全上更进一步，数据治理需要得到比现在更好的解决。

原生数据保护

除了加密之外，Hadoop 还需要遮蔽和标记化的原生方法。虽然遮蔽可以创造性地通过 UDF 或者专门的视图实现，但提供基于预设策略遮蔽在线数据的功能则会更有意义。目前有其他商业产品提供这个功能，但我们相信，Hadoop 需要一个这样的功能作为其原生功能。当前，不使用商业产品的条件下不可能支持标记化。然而标记化对于数据科学家而言很重要，尤其是因为他们可能不需要看数据的具体值，但需要保留数据的关联关系和其他静态属性以进行分析。这不可能用掩蔽实现，但可以用标记化实现。

结语

Hadoop 和大数据是令人兴奋的市场。虽然一些安全专业人员，尤其是习惯了更加统一安全功能的安全专业人员可能会对它有一些望而却步，但我们希望本书能够帮助读者理解 Hadoop 安全的现状，并体会到，即便是具有很多组件的庞大 Hadoop 集群，也能使用精心策划的安全架构进行保护。

关于作者

Ben Spivey 目前是 Cloudera 的一名解决方案架构师，负责为客户在 Hadoop 部署方面提供咨询。Ben 曾在多家世界 500 强企业工作，涉及金融服务、零售、医疗保健等多个行业。他的专长在于对客户的 Hadoop 集群进行规划、安装、配置以及安全保护。

在 Cloudera 之前，Ben 与某个国防承包商一起为美国国家安全局（NSA）工作。在此期间，Ben 的工作之一就是建立一系列应用，这些应用被集成到企业安全基础设施中以保护敏感信息。

Joey Echeverria 是 Rocana 的一位软件工程师，其工作是在 Apache Hadoop 平台建立下一代 IT 运行分析系统。Joey 还是一位 Kite SDK 贡献者，Kite SDK 是一个 Apache 许可的 Hadoop 生态系统数据 API。Joey 之前是 Cloudera 的软件工程师，在 Cloudera 期间，他为 Apache Flume、Apache Sqoop、Apache Hadoop、Apache HBase 等众多 ASF 项目做出了贡献。

关于封面

本书封面上的动物是日本獾（学名 **Meles anakuma**），鼬科。顾名思义，它是日本特有的动物，生活于本州、九州、四国和小豆岛。

与欧洲同类相比，日本獾体型较小，雄性体长约 31 英寸（79 厘米），雌性略小，平均为 28 英寸（71 厘米）。除了犬齿大小之外，雄性和雌性在体格上没有多大区别。成年日本獾的体重为 8.8 磅（4 千克）~17.6 磅（8 千克），躯干粗钝，四肢短小。其前肢拥有强大的挖土爪，而后肢较小。虽然没有欧洲獾那样独特，不过日本獾的特点是，面部有黑白相间的条纹。

日本獾在夜间活动，冬天冬眠。雌性獾 2 岁时，就会在春天求偶，并生育 2~3 只幼獾。与欧洲同类相比，日本獾更习惯独处，没有长期的配偶关系。

日本獾居住在各种树林和森林栖息地，是杂食性动物，以虫子、甲虫、浆果和柿子为食。

O'Reilly 图书封面上的很多动物都是濒危动物，它们对世界都很重要。要想详细了解如何帮助这些动物，请访问 animals.oreilly.com。

封面图片来源未知。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育 : 好书、活动每日播报
- 微博 @图灵社区 : 电子书和好文章的消息
- 微博 @图灵新知 : 图灵教育的科普小组
- 微信 图灵访谈 : ituring_interview，讲述码农精彩人生
- 微信 图灵教育 : turingbooks

091507240605ToBeReplacedWithUserId

版权信息

书名：Hadoop数据分析

作者：[美] Benjamin Bengfort Jenny Kim

译者：王纯超

ISBN：978-7-115-47964-8

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

091507240605ToBeReplacedWithUserId

版权声明
O'Reilly Media, Inc. 介绍
- 业界评论
前言
- 本书目标
- 目标读者
- 阅读方式
- 内容概述
- 编程和示例代码
  - GitHub仓库
  - 执行分布式作业
  - 使用示例代码
- 反馈及作者联系方式
- Safari® Books Online
- 联系我们
- 致谢
- 电子书
第一部分　分布式计算入门
第 1 章　数据产品时代
- 1.1　什么是数据产品
- 1.2　使用Hadoop构建大规模数据产品
  - 1.2.1　利用大型数据集
  - 1.2.2　数据产品中的Hadoop
- 1.3　数据科学流水线和Hadoop生态系统
  - 大数据工作流
- 1.4　小结
第 2 章　大数据操作系统
- 2.1　基本概念
- 2.2　Hadoop架构
  - 2.2.1　Hadoop集群
  - 2.2.2　HDFS
  - 2.2.3　YARN

2.3　使用分布式文件系统
2.3.1　基本的文件系统操作
2.3.2　HDFS文件权限
2.3.3　其他HDFS接口
2.4　使用分布式计算
2.4.1　MapReduce：函数式编程模型
2.4.2　MapReduce：集群上的实现
2.4.3　不止一个MapReduce：作业链
2.5　向YARN提交MapReduce作业
2.6　小结
第 3 章　Python 框架和 Hadoop Streaming
3.1　Hadoop Streaming
3.1.1　使用Streaming在CSV数据上运行计算
3.1.2　执行Streaming作业
3.2　Python的MapReduce框架
3.2.1　短语计数
3.2.2　其他框架
3.3　MapReduce进阶
3.3.1　combiner
3.3.2　partitioner
3.3.3　作业链
3.4　小结
第 4 章　Spark 内存计算
4.1　Spark基础
4.1.1　Spark栈
4.1.2　RDD
4.1.3　使用RDD编程
4.2　基于PySpark的交互性Spark
4.3　编写Spark应用程序
使用Spark可视化航班延误
4.4　小结
第 5 章　分布式分析和模式

5.1　键计算
5.1.1　复合键
5.1.2　键空间模式
5.1.3　pair与stripe
5.2　设计模式
5.2.1　概要
5.2.2　索引
5.2.3　过滤
5.3　迈向最后一英里分析
5.3.1　模型拟合
5.3.2　模型验证
5.4　小结
第二部分　大数据科学的工作流和工具
第 6 章　数据挖掘和数据仓储
6.1　Hive结构化数据查询
6.1.1　Hive命令行接口（CLI）
6.1.2　Hive查询语言
6.1.3　Hive数据分析
6.2　HBase
6.2.1　NoSQL与列式数据库
6.2.2　HBase实时分析
6.3　小结
第 7 章　数据采集
7.1　使用Sqoop导入关系数据
7.1.1　从MySQL导入HDFS
7.1.2　从MySQL导入Hive
7.1.3　从MySQL导入HBase
7.2　使用Flume获取流式数据
7.2.1　Flume数据流
7.2.2　使用Flume获取产品印象数据
7.3　小结
第 8 章　使用高级 API 进行分析

8.1 Pig
8.1.1 Pig Latin
8.1.2 数据类型
8.1.3 关系运算符
8.1.4 用户定义函数
8.1.5 Pig小结
8.2 Spark高级API
8.2.1 Spark SQL
8.2.2 DataFrame
8.3 小结
第 9 章 机器学习
9.1 使用Spark进行可扩展的机器学习
9.1.1 协同过滤
9.1.2 分类
9.1.3 聚类
9.2 小结
第 10 章 总结：分布式数据科学实战
10.1 数据产品生命周期
10.1.1 数据湖泊
10.1.2 数据采集
10.1.3 计算数据存储
10.2 机器学习生命周期
10.3 小结
附录 A 创建 Hadoop 伪分布式开发环境
A.1 快速上手
A.2 设置Linux环境
A.2.1 创建Hadoop用户
A.2.2 配置SSH
A.2.3 安装Java
A.2.4 禁用IPv6
A.3 安装Hadoop
A.3.1 解压

A.3.2　环境
A.3.3　Hadoop配置
A.3.4　格式化NameNode
A.3.5　启动Hadoop
A.3.6　重启Hadoop
附录 B　安装 Hadoop 生态系统产品
B.1　打包的Hadoop发行版
B.2　自己安装Apache Hadoop生态系统产品
B.2.1　基本安装和配置步骤
B.2.2　Sqoop特定配置
B.2.3　Hive特定配置
B.2.4　HBase特定配置
B.2.5　安装Spark
术语表
关于作者
关于封面

版权声明

© 2016 by Jenny Kim and Benjamin Bengfort

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2018. Authorized translation of the English edition, 2016 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2016。

简体中文版由人民邮电出版社出版，2018。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O'Reilly Media, Inc. 介绍

O'Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O'Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly 现在还

将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O'Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar 博客有口皆碑。”

——*Wired*

“O'Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O'Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O'Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

前言

大数据已经成为一个流行词。人们用它来描述数据驱动型应用程序中的那些令人兴奋的新工具和新技术。这些应用程序正为我们带来崭新的计算方式。令统计学家懊恼的是，这一词语似乎被随意使用，其范围甚至包括在大型数据集上使用众所周知的统计技术进行预测。虽然大数据已经成为流行词，但事实上，现代分布式计算技术能分析的数据集远比过去那些“典型”方式能应对的数据集大得多，结果也更令人

震撼。

然而，单纯的分布式计算并不等于数据科学。互联网带来了快速增长的数据集，这些数据集又能驱动预测模型（“更多的数据优于更好的算法”1），数据产品也因此成为了一种新型的经济范式。为大型跨域异构数据集建模所取得的巨大成功（例如 Nate Silver 通过大数据技术像使用魔法一般预测了 2008 年的美国大选结果），使很多人认识到了数据科学的价值，也为这个领域吸引了大量从业者。

> 1Anand Rajaraman,“More data usually beats better algorithms”(http://anand.typepad.com/datawocky/2008/03/more-data-usual.html), Datawocky, March 24, 2008.

通过提供分布式数据存储和并行计算框架，Hadoop 已经从集群计算的抽象演变成了大数据操作系统。Spark 正是基于这一理念构建的，它使数据科学家能更轻松地使用集群计算。然而，不了解分布式计算的数据科学家和分析人员可能会觉得这些工具是面向程序员的，而不是面向分析人员的。这是因为，我们需要从根本上转变管理数据和计算数据的思维方式，这样才能从串行模式转换到并行模式。

本书旨在通过可读且直观的方式介绍集群计算和分析，帮助数据科学家完成这一思维转换。我们将针对数据分析介绍分布式计算涉及的大量概念、工具和技术，为深入了解特定领域铺平道路。

本书目标

本书不会详细讲解 Hadoop（推荐 Tom White 的《Hadoop 权威指南》），也不是 Spark 入门资料（推荐 Holden Karau 等人所著的《Spark 快速大数据分析》2），当然更不是为了教你如何进行分布式计算。本书将纵览 Hadoop 生态系统和分布式计算，旨在武装数据科学家、统计学家、程序员和对 Hadoop 感兴趣（但是对 Hadoop 的了解十分有限）的人。希望本书能成为你深入 Hadoop 世界的向导，助你找到最感兴趣的工具和技术。这可能是 Spark、Hive、机器学习、ETL（抽取、转换和加载）操作、关系数据库或者众多与集群计算相关的主题之一。

> 2本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1558。——编者注

目标读者

人们经常把数据科学与大数据混为一谈。虽然为了达到良好的泛化效果，许多机器学习模型确实需要大型数据集，但即使是小型数据集也能支持模式识别。因此，数据科学软件类的图书大多关注易于在一台机器（尤其是内存容量多达几吉字节的机器）上分析的数据集。尽管大数据和数据科学非常适合协同工作，但是直到今天，与计算相关的图书还是将它们分开讨论。

本书以数据科学家为目标读者，旨在弥补这一隔阂。它将以数据科学的视角介绍 Hadoop 集群计算和分析。本书的关注点不是部署、运维或软件开发，而是常用分析、数据仓储技术和高阶数据流。

那么，什么人算是数据科学家呢？本书所说的数据科学家是指具有高超统计技能的软件开发人员，或者具有强大软件开发能力的统计学家。通常情况下，数据团队由三类数据科学家组成，分别是数据工程师、数据分析师和领域专家。

数据工程师指能构建或者使用高级计算系统的程序员或者计算机科学家。他们通常使用 Python、Java 或者 Scala 编程，熟悉 Linux、服务器、网络、数据库和应用程序部署。如果你是数据工程师，本书假设你能适应多进程编程、数据整理和数值计算。希望你在阅读本书后，能更了解如何在集群上部署应用程序，学会如何处理比单机在足够时间内能处理的数据集还要大得多的数据集。

数据分析师主要关注统计建模和探索性数据分析。在日常工作中，他们通常使用 R、Python 或者 Julia，熟悉数据挖掘和机器学习技术，比如回归、聚类和分类问题。数据分析师很可能通过采样处理过更大的数据集。我们将在本书中展示数据统计技术，处理比以往获取的数据量大得多的数据，从而构建预测能力既有广度又有深度的模型。

领域专家是团队里富有影响力、面向业务的成员。他们深入了解数据类型和所碰到的问题，理解数据带来的特定挑战，并寻求通过更好的方式利用数据应对新挑战。希望本书能够为他们提供一些业务决策思路，让当前的数据流更加灵活，并帮助他们理解怎样使用通用的计算框架来应对特定的领域挑战。

阅读方式

至今为止，Hadoop 已经有十多年的历史了，就技术而言，这已经是很长一段时间了。然而，摩尔定律仍然没有慢下来。10 年前，在数据中心使用廉价的机器集群远比为超级计算机编程简单。但现在，同

样的廉价服务器要比以前强大约 32 倍，内存计算的开销也降低了很多。Hadoop 成为了大数据的操作系统，支持图形处理、类 SQL 查询和流处理等多种计算框架。但这也给想要学习 Hadoop 的人带来了巨大的挑战——该从何学起？

本书篇幅简短的原因只有一个——想要尽可能简洁地覆盖多个方面。我们希望你通过两种方式阅读本书：一是快速通读全书，对 Hadoop 和分布式数据分析有大致了解；二是选择感兴趣的章节深入学习。本书以易懂为目的。我们通过简单的代码示例进行讲解，不一定需要你亲自实现和运行代码。本书是 Hadoop 和 Spark 领域的指导手册，对分析人员尤其如此。

内容概述

本书旨在带领你了解 Hadoop 生态系统，书中内容分为两部分：第一部分（第 1 章至第 5 章）宏观地介绍分布式计算，讨论如何在集群上运行计算；第二部分（第 6 章至第 10 章）侧重于介绍数据科学家应该具体了解的工具和技术，意在为各种分析和大规模数据管理提供动力。（第 5 章将从对分布式计算的讨论过渡到更加具体的工具和大数据科学流水线的实现。）每章的内容概述如下。

第 **1** 章数据产品时代

　　介绍大数据和数据科学的结晶——数据产品，讨论创建数据产品背后的流程，说明数据分析的串行模型如何与分布式计算相契合。

第 **2** 章大数据操作系统

　　概述 Hadoop 背后的核心概念，讲解为何集群计算既有益又复杂；主要着眼于 YARN 和 HDFS，详细讨论 Hadoop 体系架构，讲解与分布式存储系统的交互，为分析大型数据集作准备。

第 **3** 章 **Python** 框架和 **Hadoop Streaming**

　　介绍分布式计算的基本编程抽象 MapReduce。然而，MapReduce 的 API 是用 Java 编写的，这不是一种在数据科学家间流行的编程语言。因此，这一章专注于介绍如何通过 Hadoop Streaming 使用 Python 编写 MapReduce 作业。

第 **4** 章 **Spark** 内存计算

虽然理解 MapReduce 对理解分布式计算和编写高性能的批处理作业（如 ETL）十分重要，但是 Hadoop 集群上的日常交互和分析却通常都是使用 Spark 完成的。这一章将介绍 Spark，以及如何使用 Python 编写 Spark 应用程序，并通过 PySpark 以交互方式在 YARN 上运行，或者在集群模式下运行。

第 **5** 章分布式分析和模式

通过展示设计模式和并行分析算法，从实践的角度研究怎样编写分布式数据分析作业。开始阅读这一章之前，你应该已经了解编写 Spark 和 MapReduce 作业的原理。读完这一章，你应该能轻松实现它们。

第 **6** 章数据挖掘和数据仓储

介绍分布式环境下的数据管理、数据挖掘和数据仓储，特别是与传统数据库系统密切相关的方面。这一章重点介绍 Hive 和 HBase，它们分别是 Hadoop 最流行的基于 SQL 的查询引擎和 NoSQL 数据库。数据整理是数据科学流水线的第二步，但是数据需要被采集到某处。这一章还将探索怎样管理大型数据集。

第 **7** 章数据采集

考虑到数据的容量和速度，如何将数据导入分布式系统并用于计算可能才是最大的挑战之一。这一章将研究从关系数据库获取数据的批量加载工具 Sqoop 以及更灵活的 Apache Flume，后者用于获取日志和来自网络的其他非结构化数据。

第 **8** 章使用高级 **API** 进行分析

研究用于编写复杂 Hadoop 和 Spark 应用程序的高阶工具，尤其是 Apache Pig 和 Spark 的 DataFrame API。第一部分将讨论 MapReduce 和 Spark 分布式作业的实现过程，以及怎样从数据流的角度看待算法和数据流水线。Pig 让你无须使用 MapReduce 实现底层细节，从而能更轻松地描述数据流。Spark 提供了多个集成模块，能无缝结合过程式处理与关系查询，为强大的分析定制打开了大门。

第 **9** 章机器学习

大数据的多数益处都是在机器学习中得以实现的——更加广泛的特征和输入空间让模式识别技术更加有效和个性化。这一章将介绍分类、聚类和协同过滤，但并不会详细讨论建模，而是使用 Spark 的

MLlib 让你上手可扩展机器学习技术。

第 **10** 章总结：分布式数据科学实战

完整呈现分布式数据科学，把前面章节中单独讨论的工具与技术结合起来。数据科学不是单一的活动，而是一个生命周期，涉及数据的采集、整理、建模、计算和操作化。这一章将从整体上讨论分布式数据科学的架构和工作流。

附录 **A** 创建 **Hadoop** 伪分布式开发环境

附录 A 将指导你在本机上搭建一个开发环境，从而编写分布式作业。如果你没有集群可用，附录 A 是运行本书示例至关重要的准备工作。

附录 **B** 安装 **Hadoop** 生态系统产品

附录 B 是附录 A 的延伸，将提供本书讨论的众多生态系统工具和产品的安装指导。尽管附录 A 提供了安装服务的常用方法，但附录 B 专门为安装服务（用来运行书中示例，你在阅读的过程中会遇到它们）的过程中会遇到的问题提供指导。

你看，这么薄的一本书却涵盖了这么多主题。希望以上这些内容足以吸引你继续阅读下去。

编程和示例代码

随着 Hadoop 的分布式计算变得更加成熟和集成化，并行计算正在向更丰富的分析体验转变。例如，大数据生态系统的最新成员 Spark 提供了 4 种语言的编程 API，更方便那些习惯于使用数据框、交互式 notebook 和解释型语言的数据科学家使用。Hive 和 SparkSQL 以 SQL 语法形式提供了另外一种为人们所熟知的领域专用语言（domain-specific language，DSL），专门针对分布式集群上的数据查询。

因为本书的目标读者是数据科学家，所以我们大多选择使用 Python 来实现示例。Python 是通用的编程语言，拥有丰富的分析包（例如 Pandas 和 Scikit-Learn），在数据科学领域占有一席之地。不幸的是，Hadoop 的主要 API 通常都是以 Java 编写的，那些 Python 示例让我们大费周章，但是大多数时候，我们会用更实际的方式来阐明思想。因此，本书中的代码要么是使用 Python 和 Hadoop Streaming

的 MapReduce，要么是使用 PySpark API 的 Spark 代码，或者是讨论 Hive、Spark SQL 时的 SQL 代码。希望这能让更多读者感觉简明易懂。

GitHub仓库

你可以在我们的 GitHub 仓库（https://github.com/bbengfort/hadoop-fundamentals）找到本书完整且可执行的示例代码。这个仓库也包含了我们的 Hadoop 视频教程“Hadoop Fundamentals for Data Scientists”[3] 的代码。

> [3]http://shop.oreilly.com/product/0636920035183.do.

为了在纸质版中呈现代码并更清楚地解释过程，我们走了捷径，省略了代码的细节。例如，通常都省略了 `import` 语句——这意味着简单的复制粘贴无法奏效。然而，你可以使用仓库中的例子，它们是完整且可执行的代码，并附有相应的注释。

但要注意，仓库是持续更新的，可以查阅 README 文件以了解更新情况。你当然可以 fork 仓库，更改代码以在你自己的环境中运行——我们强烈推荐你这样做！

执行分布式作业

Hadoop 开发人员通常在“伪分布式模式”下使用“单节点集群”进行开发任务。该集群通常是一个运行着虚拟服务器环境的虚拟机，环境中运行着多个 Hadoop 守护进程。你可以在主开发工具里使用 SSH 访问该虚拟机，就像访问 Hadoop 集群一样。为了创建虚拟环境，你需要某种虚拟化软件，例如 VirtualBox（https://www.virtualbox.org）、VMWare（http://www.vmware.com/products/desktop-virtualization）或者 Parallels（http://www.parallels.com）。

附录 A 讨论怎样设置以伪分布式模式运行 Hadoop、Hive 和 Spark 的 Ubuntu x64 虚拟机。你也可以使用一些 Hadoop 发行版（例如 Cloudera 和 Hortonworks）提供的预先配置好的虚拟环境。如果你有想用的虚拟机环境，那么我们建议你下载它。如果你想了解更多的 Hadoop 操作，就请自己配置吧！

还有一点，因为 Hadoop 集群是在开源软件上运行的，所以你需要了

解 Linux 和命令行。本书讨论的虚拟机通常都是通过命令行访问的，书中的许多例子都描述了通过命令行与 Hadoop、Spark、Hive 和其他工具交互的过程。命令行是分析人员不愿使用这些工具的一个主要原因。然而，学习命令行对你大有帮助，它也并不可怕。我们建议你学习一下！

使用示例代码

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无须联系我们获得许可。比如，用本书的几个代码片段写一个程序就无须获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无须获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN，比如“*Data Analytics with Hadoop* by Benjamin Bengfort and Jenny Kim (O'Reilly). Copyright 2016 Benjamin Bengfort and Jenny Kim, 978-1491-91370-3”。

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

反馈及作者联系方式

关于本书的评论和技术性问题，请发送电子邮件至 bookquestions@oreilly.com。

工具和技术的变化非常快，在大数据领域尤其如此。不幸的是，很难保证一本书（特别是纸质版）时刻跟上潮流。我们希望本书能够在未来继续为你服务，但如果你发现有变更让书中的示例无法运行或导致代码问题，请联系我们。

如果有关于代码或示例的问题，请在 GitHub 上（https://github.com/bbengfort/hadoop-fundamentals/issues/）提交问题，这是与我们联系的最佳方式。你也可以发送电子邮件到 hadoopfundamentals@gmail.com，我们会尽快回复。非常感谢你提供积极且具有建设性的反馈！

Safari® Books Online



Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几百家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。美国：

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）

奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。4 本书的网页是：

> 4也可以通过图灵社区获取相关信息：http://www.ituring.com.cn/book/1944。——编者注

http://shop.oreilly.com/product/0636920035275.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

我们要感谢本书的审校者，你们在漫长的写作过程中持续提出颇具建设性的反馈和批评。感谢 Marck Vaisman，你从向数据科学家讲授 Hadoop 的角度阅读本书；特别感谢 Konstantinos Xirogiannopoulos 在忙碌的研究之余，志愿向我们提供清晰、有益且积极的评论，我们很高兴收到这样的评论。

我们还要感谢 O'Reilly 每位富有耐心、坚持不懈的编辑。在本书写作之初，我们走了一些弯路，是 Meghan Blanchette 带领我们一路走了过来，她一直支持着我们。很遗憾，在本书还未截稿时，她已离开 O'Reilly 去追求更好的事业。当 Nicole Tache 接替她并成功将我们带

回正轨时，我们特别高兴。Nicole 引导我们完成写作，没有她，就没有本书。她有一项特殊的本领，总能在关键时刻发来让人看了就舒心的邮件，让工作如期完成。与 O'Reilly 每个人的合作都很愉快，特别是 Marie Beaugureau、Amy Jollymore、Ben Lorica 和 Mike Loukides，感谢你们给予的建议和鼓励。

在华盛顿，District Data Labs 的工作人员给予了我们巨大的支持。我们要特别提到 Tony Ojeda、Rebecca Bilbro、Allen Leis 和 Selma Gomez Orr，你们以各种方式支持本书，包括第一个购买早期版本、提供反馈、审查代码并询问完成时间，这都鼓励我们专心写作！

如果没有 Hadoop 社区中各位了不起的贡献，本书就不可能诞生，Jenny 更是有幸每天在 Cloudera 与这些志同道合者一起工作。特别感谢 Hue 团队，他们为提供最好的 Hadoop 用户体验所做出的贡献和表现出的热情既超乎寻常又鼓舞人心。

感谢我们的家人，特别是 Benjamin 的父母 Randy Bengfort 和 Lily Bengfort，以及 Jenny 的父母 Wung Kim 和 Namoak Kim，感谢你们一直以来的鼓励、爱与支持。我们的父母向我们灌输了相互学习和探索的热情，因此我们钻研了许多方面。他们使我们拥有坚韧的精神和恒心，总能让我们找到实现目标的方法。

最后，感谢我们的伴侣 Patrick 和 Jacquelyn，感谢你们一直支持我们。我们中的谁可能说过："再写一本书，我的婚姻就要结束了。"诚然，在写作的最后阶段，他们都不愿意听到我们仍然在努力。但如果没有他们，我们的书就无法写成（"婚姻不在，书也就不在了"）。在我们通过视频电话商定细节和重写时，Patrick 和 Jacquelyn 总是和颜悦色。他们甚至阅读了部分内容，提供建议，在各方面都提供了帮助。在这之前，我们都没有写过书，也不知道会面临什么问题。现在我们知道了，也很高兴他们一直在身边支持我们。

电子书

扫描如下二维码，即可购买本书电子版。



第一部分　分布式计算入门

本书第一部分将介绍如何使用 Hadoop 进行大数据的分布式计算：第 1 章引入数据产品的构建对分布式计算的需求，并讨论在数据科学领域使用 Hadoop 的主要工作流和时机；第 2 章深入讲解分布式存储和分布式计算在需求方面的技术细节，并解释 Hadoop 成为大数据操作系统的原因；第 3 章和第 4 章分别介绍使用 MapReduce 和 Spark 的分布式编程；第 5 章从数据科学家分析大型数据集的角度探讨 MapReduce 和 Spark 中常见的计算和模式。

第 1 章　数据产品时代

我们生活在一场信息革命之中。和任何经济革命一样，它对社会、学术界和商界都造成了极具变革性的影响。眼前这场革命由网络通信系统和互联网驱动，其独特之处在于创造了大量有价值的新材料——数据，并将所有人转变成了消费者和生产者。每天都有海量的数据生成，数据也日益影响着生活的各个方面，从吃的食物，到社交活动，再到工作和娱乐的方式。同样，我们也对产品和服务有了合理的期

望，比如高度个性化，或能针对我们的身体、生活和职业进行优化。这为一项崭新的信息技术创造了市场——数据产品。

过剩的数据集与机器学习算法快速敏捷地结合，这不仅改变了人们与日常事物的交互方式，也改变了彼此打交道的方式，因为这一结合经常带来立竿见影且富有新意的成果。的确，大量的模型和数据源似乎带来了无穷无尽的创新，围绕“大数据”这一热词的趋势正与此有关。

数据产品通过数据科学工作流创建；具体来说，是将模型（通常是预测性的或推断性的）应用于特定领域的数据集。虽然创新的潜力是巨大的，但是发现数据源并正确建模或挖掘模式需要科学性或实验性的思维模式，而程序员和分析人员通常不具备这一能力。也正是出于这个原因，雇用博士确实省时省力——他们经过必要的分析和实验训练，结合编程，基本立即就能成为数据科学专业人才。当然，我们不可能都是博士。因此，本书提出了一个将 Hadoop 用于大规模数据科学的教学模型，并且将其作为构建应用程序的基础，这些应用程序正是（或可以成为）数据产品。

1.1　什么是数据产品

这个问题的传统答案通常是：“任何将数据和算法结合起来的应用程序。”[1] 但坦白说，如果你编写的软件没有将数据与算法结合在一起，那么你在做什么呢？毕竟数据可是编程界的“货币”! 具体来说，数据产品就是数据与用于推断或预测的统计算法的结合。许多数据科学家也是统计学家，统计方法论是数据科学的核心。

> 1Hillary Mason and Chris Wiggins,“A Taxonomy of Data Science”(http://www.dataists.com/2010/09/a-taxonomy-of-data-science/), Dataists, September 25, 2010.

根据这个定义，Amazon 的推荐系统就是一个数据产品。Amazon 会检查你购买的商品，并根据其他用户类似的购买行为作出推荐。在这种情况下，将订单历史数据与推荐算法相结合，就能预测你将来可能购买什么。Facebook 的“你可能认识的人”也是一个数据产品，因为它“基于共同的好友、工作、教育信息以及许多其他因素向你推荐好友”——这本质上是结合社交网络数据与图算法来推断社区成员。

这两个产品分别在零售业和社交网络领域具有革命性意义，但它们看上去与其他 Web 应用程序没什么不同。的确，简单地将数据产品定义为数据与统计算法的结合，似乎将其限制为了单一的软件类型（如

Web 应用程序），这很难成为一股革命性的经济力量。虽然可以说 Google 或其他公司是庞大的经济力量，但是仅将收集庞大 HTML 语料库的 Web 爬虫与 PageRank 算法结合却不能创造数据经济。众所周知，搜索在经济活动中起到了重要作用，因此上述定义一定有缺失。

Mike Loukides 认为，数据产品不仅仅是“数据驱动的应用程序”。虽然博客、电子商务平台以及大多数 Web 应用程序和移动应用程序都依赖于数据库和数据服务（如 RESTful API），但它们只使用数据，其本身并不构成数据产品。他对数据产品的定义如下。2

> 2Mike Loukides,“What is Data Science?”(https://www.oreilly.com/ideas/what-is-data-science), O'Reilly Radar, June 2, 2010.

> 数据应用程序从数据本身获取价值，然后创造更多数据。它不仅仅是带有数据的应用程序，而是数据产品。

这是一场革命，数据产品则是经济引擎。它从数据中获取价值，作为回报，再产生更多的数据和价值。它产生的数据可为自己添加燃料（我们终于实现了永动机），或者催生其他数据产品，这些数据产品从生成的数据中获取价值。正是这一过程带来了信息过剩和信息革命。更重要的是，这种生成效应让我们通过数据过上了更好的生活，因为更多的数据产品意味着更多的数据，更多的数据意味着更多的数据产品，循环往复。

有了这个更具体的定义，我们就可以更进一步，将数据产品描述为一个从数据中学习、自适应并且广泛适用的系统。根据这个定义，Nest 恒温器算是数据产品：它从传感器数据中获取价值，决定制热或制冷，并且收集新的传感器数据以检验调节效果。由斯坦福大学的无人驾驶团队研制的无人驾驶汽车也属于这一类。该团队通过算法实现机器视觉并模拟驾驶员行为，因此车辆在行驶时能产生更多导航数据和传感器数据，这些数据可用于改进驾驶平台。此外，由 Fitbit、Withings 和许多其他公司发起的“量化自我”产品意味着数据可以影响人类行为；智能电网意味着数据会影响你的效能。

数据产品是自适应且广泛适用的经济引擎。它从数据中获取价值，并通过影响人类行为或通过基于新数据的推断或预测产生更多数据。数据产品不只是 Web 应用程序，它正迅速成为现代社会几乎每一个经济活动领域的重要组成部分。数据产品能够发现人类活动中的个体模式，所以它能推动决策，由它引发的行动和影响也会被记录为新的数

据。

1.2　使用Hadoop构建大规模数据产品

Josh Wills 经常被引用的推文 3 给我们提供了以下定义。

> 3https://twitter.com/josh_wills/status/198093512149958656.

> 数据科学家（名词）：指比所有软件工程师更擅长统计学，并且比所有统计学家更擅长软件工程的人。

当然，这与数据产品仅仅是数据与统计算法的结合这一想法十分吻合。软件工程和统计学知识都是数据科学的基础。然而，在一个需要产品从数据中获取价值并产生新数据的经济体系中，构建数据产品其实就是数据科学家的工作。

Harlan Harris 提供了有关数据产品的更多细节 4：它们建立在数据、领域知识、软件工程和分析技术的交叉点上。由于数据产品是系统，因此构建它们需要工程技能，通常是软件工程方面的技能；由于它们由数据驱动，因此拥有数据是必要条件；领域知识和分析技术是用于构建数据引擎的工具，通常通过实验完成，因此是数据科学的“科学”部分。

> 4Harlan Harris,“What Is a Data Product?”, Analytics 2014 Blog, March 31, 2014.

由于需要使用实验方法学，因此大多数数据科学家会采用典型的分析工作流：采集 → 整理 → 建模 → 报告和可视化。然而，这种所谓的数据科学流水线完全由人力驱动，再辅以脚本语言（如 R 和 Python）的使用。流水线的每一个环节都需要人类的知识和分析技能，意在产生独特且不可泛化的结果。虽然这个流水线是很好的统计和分析基础框架，但它不能满足构建数据产品的需求，特别是当想从中获取价值的目标数据大到无法在一台笔记本电脑上处理时。随着数据越来越多、越来越多变、产生的速度越来越快，自动获取有用信息而无须人工干预的工具也变得越来越重要。

1.2.1　利用大型数据集

直觉告诉我们，观测越多，数据就越多——这真让人喜忧参半。人类拥有发现大规模模式的卓越能力（我们以森林和林中空地作为隐

喻）。理解数据的认知过程涉及概览数据，深入研究具体层面的细节，然后再回到概览角度。这个过程中的细节并不一定可靠，因为细粒度（隐喻中的叶子、分枝或单棵树木）会限制我们的理解能力。多数数据既可能是模式和信号，也可能是噪声和干扰。

通过聚合和索引描述数据，或者直接对数据建模，统计方法使我们能够处理掺杂着噪声和信号的数据。虽然这些技术能帮助我们理解数据，但是它们以牺牲计算粒度为代价，例如有意义的罕见事件可能会被模型排除。兼顾罕见事件的统计技术能利用计算机同时跟踪多个数据点，但也需要更多的计算资源。因此，传统的统计方法会对较大的数据集采取抽样方法，用较小的数据子集替代总体。样本越大，模型包括罕见事件且能将其捕获的可能性就越大。

随着收集数据的能力越来越高，我们对通用性也有了更大的需求。过去十年间，由于数据和机器学习算法的紧密结合，新颖的成果纷纷问世，数据科学得到了空前的发展。智能电网、“量化自我”、移动技术、传感器和互联家庭要求我们应用个性化的统计推断。规模不仅与数据量有关，也与需要探索多少方面有关——就好像森林中的每棵树一样。

Google 的两篇论文描述了一个完整的分布式计算系统；Hadoop 是其开源实现，它将我们带入了大数据时代。然而，分布式计算和分布式数据库系统并不是新的话题。在那两篇论文发表之前，与 Hadoop 的计算能力相当的数据仓库系统就早已存在于工业界和学术界。Hadoop 之所以与众不同，一方面是因为数据处理能带来经济效益，另一方面是因为 Hadoop 是一个平台。但是真正使 Hadoop 独树一帜的原因其实是它出现的时机——恰恰在一个需要大规模数据分析解决方案的时刻，它问世了。而且它不仅能分析总体的统计数据，还能获得个体级别的通用性和洞察力。

1.2.2　数据产品中的Hadoop

一开始，Hadoop 的使用者是那些面临大数据挑战的大公司，比如 Google、Facebook 和 Yahoo。然而，Hadoop 之所以这么重要，以及促使你拿起本书的原因，恰恰是因为面临数据挑战的不再只是科技巨擘。大大小小的商业机构和政府机构——从企业到创业公司，再到联邦机构和市政府，甚至是每个人，都面临着数据挑战。计算资源变得廉价且唾手可得——就像 PC 时代的黑客在车库里使用手边的电子产品搞创新，现在的创业公司使用 10 节点～ 20 节点的小集群在数据探索上搞创新。云计算资源（如 Amazon EC2 和 Google Compute

Engine）使数据科学家可以及时、按需地访问大规模集群，而且成本较低，也无须进行数据中心管理。Hadoop 使大数据计算更贴近大众，也更平易近人，下面的例子说明了这一点。

2011 年，Lady Gaga 发行了她的专辑 *Born This Way*，这个事件为社交媒体带来了约 1.3 万亿条信息，包括点赞、推文、图像和视频。Lady Gaga 的经纪人 Troy Carter 马上发现了一个将粉丝聚集起来的机会。经过大量的数据挖掘工作，他成功将 Twitter 和 Facebook 上的数百万粉丝聚集到了 LittleMonsters.com 这个只针对 Lady Gaga 的小社交网络中。该网站的成功促使 Backplane（现在叫 Place）诞生，这是一个用于生成和管理由小型社区驱动的社交网络的工具。

2015 年，纽约市警察局安装了一个价值 150 万美元的声学传感器网络，名叫 ShotSpotter。该系统能够检测与爆炸或枪击相关的脉冲声，使应急响应人员能够快速响应在布朗克斯区发生的事件。重要的是，这个系统还很智能，可以预测是否会发生后续的枪击事件及其大致位置。ShotSpotter 系统发现，自 2009 年以来，有超过 75%的枪击事件没有报告给警察。

"量化自我"运动越来越受欢迎，各家公司也一直努力在消费者中广泛普及可穿戴技术设备、个人数据收集设备，甚至是基因测序仪器。2012 年，美国的《平价医疗法案》规定健康计划对电子病历实施标准化、安全、保密的共享方法。互联家庭、移动设备以及其他个人传感器每天都在产生大量个人数据，这引发了人们对隐私的关注。2015 年，英国研究人员创建了 *Hub of All Things*（HAT）。这是一项个性化的数据集合技术，用于处理"谁拥有你的数据"这一问题，并为个人数据的聚合提供技术解决方案。

传统上，大规模的个人数据分析一直属于社交网络的范畴，如 Facebook 和 Twitter。但幸好有了 Place，大型社交网络现在成为了个人品牌和艺术家的诞生之地。每个城市面临的数据挑战都不一样，尽管针对典型城市的泛化可以满足许多分析的需求，但是新的数据挑战仍然不断出现，对每个城市分别进行研究势在必行。（比如工业、航运或天气对声学传感器网络有什么影响？）怎样使技术为消费者提供价值，在使用他们的个人医疗记录时不侵犯他们的隐私，避免与其他人的记录聚合？怎样使个人医疗诊断数据挖掘变得更安全？

数据产品的出现正是为了切实回答这些问题。Place、ShotSpotter、"量化自我"产品和 HAT 等通过提供应用程序平台和决策资源供人们采取行动，从数据中获取价值并产生新数据。它们提供的价值是明确的，但要处理数万亿的点赞数据和数百万个麦克风生成

的大量数据集，或者我们每天生成的海量个人数据，传统的软件开发工作流无法应对这一挑战。大数据工作流和 Hadoop 使这些应用程序成为可能并且可个性化。

1.3 数据科学流水线和Hadoop生态系统

数据科学流水线是一种教学模型，用于教授对数据进行全面统计分析所需的工作流，如图 1-1 所示。在每个环节中，分析人员要转换初始数据集，然后从各种数据源增强或采集数据，再通过描述性或推断性的统计方法将数据整理为可以计算的正常形式，最后通过可视化或报告的形式生成结果。这些分析过程通常用于回答特定问题，或用于调查数据与某些业务实践间的关系，以进行验证或决策。

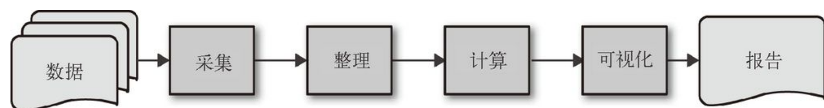


图 **1-1**：数据科学流水线

这个原始的工作流模型引领了大多数早期的数据科学思想。最初关于数据科学应用程序的讨论围绕着如何创建有意义的信息可视化——这也许令人意外，主要是因为这个工作流旨在生成帮助人们进行决策的依据。通过对大型数据集的聚合、描述和建模，人们能够更好地根据模式（而不是单个数据点）作出判断。数据可视化是新生的数据产品，它们从数据中产生价值，帮助人们基于学习到的内容采取行动，然后再从这些行动中生成新数据。

然而，面对呈指数增长的数据量和数据增长速度，这种以人力驱动的模型并不是一个可扩展的解决方案，这也正是许多企业都为之抓狂的原因。根据预测，到 2020 年，我们每年生成和复制的数据将达到 44ZB，即 44 万亿 GB。5 即使实际规模只达到预测规模的一小部分，手动的数据准备和挖掘方法也根本无法及时提供有意义的信息。

> 5EMC Digital Universe with Research & Analysis by IDC,“The Digital Universe of Opportunities”(https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm), April 2014.

除了规模上的局限，这种以人为中心的单向工作流也不能有效地设计能够学习的自适应系统。机器学习算法已经广泛应用于学术界之外，非常符合数据产品的定义。因为模型会拟合现有的数据集，所以这些

类型的算法可以从数据中获取价值，然后通过对新的观察值作出预测来产生新的数据。

如果要创建一个框架，支持构建可扩展和可自动化的解决方案，从而能解释数据和生成有用的信息，就必须修改数据科学流水线，使其包含机器学习方法的反馈循环。

大数据工作流

考虑到可扩展性和自动化的目标，我们可以将人力驱动的数据科学流水线重构为包括采集、分段、计算和工作流管理这 4 个主要阶段的迭代模型（如图 1-2 所示）。与数据科学流水线一样，这种模型其实就是采集原始数据并将其转换为有用的信息。关键的区别在于，数据产品流水线是在操作化和自动化工作流的步骤中构建起来的。通过将采集、分段和计算这 3 个步骤转换为自动化工作流，最终产生可重用的数据产品。工作流管理步骤还引入了反馈流机制，来自其中一个作业执行的输出可以自动作为下一次迭代的数据输入，因此为机器学习应用程序提供了必要的自适应框架。

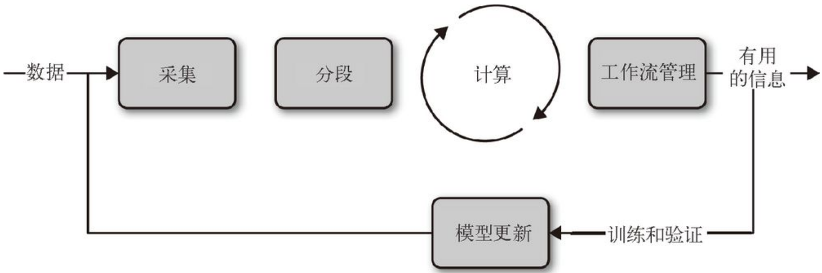


图 **1-2**：大数据流水线

采集阶段既是模型的初始化阶段，也是用户和模型之间的应用交互阶段。在初始化期间，用户指定数据源的位置或标注数据（另一种数据采集形式）；在交互期间，用户消费模型的预测结果并提供用于巩固模型的反馈。

分段阶段是转换数据的阶段，使其变为可消费的形式并存储起来，从而能够用于处理。本阶段还负责数据的归一化和标准化，以及一些计算数据存储中的数据管理工作。

计算阶段是真正“干活”的阶段，主要负责挖掘数据以获取有用的信息，执行聚合或报告，构建用于推荐、聚类或分类的机器学习模型。

工作流管理阶段执行抽象、编排和自动化任务，使工作流的各步骤可用于生产环境。此步骤应能产生自动按需运行的应用程序、作业或脚本。

Hadoop 已经演变成了包含各种工具的生态系统，可以实现上述流水线的部分环节。例如，Sqoop 和 Kafka 可用于数据采集，支持将关系数据库导入 Hadoop 或分布式消息队列，以进行按需处理。在 Hadoop 中，像 Hive 和 HBase 之类的数据仓库提供了大规模的数据管理机会；Spark 的 GraphX、MLlib 或 Mahout 库提供了分析包，供大规模计算和验证使用。在本书中，我们将探索 Hadoop 生态系统的许多组件，并了解它们如何融入整个大数据流水线。

1.4　小结

在过去十年间，关于“什么是数据科学”的讨论发生了巨大变化——从纯分析到与可视化相关的方法，再到如今数据产品的创建。数据产品是使用数据训练、自适应且广泛适用的经济引擎，从数据中获取价值并产生新的数据。数据产品引发了一次信息经济革命，改变了小企业、技术创业公司、大型组织甚至政府机构看待其数据的方式。

本章描述了数据科学流水线原始教学模型的一个改良版本，并提出了数据产品流水线。数据产品流水线是迭代的，包括两个阶段：构建阶段和运行阶段（包括 4 个阶段：交互、数据、存储和计算）。这种架构可以有条不紊地执行大规模的数据分析，保留了实验、人与数据产品间的交互。而且当围绕数据产品构建的应用程序很大时，它还能支持部分环节的自动化。希望这个流水线可以帮你了解数据产品生命周期的大体框架，也能成为探索更多创新项目的基石。

因为本书是从数据科学家的角度探讨分布式计算和 Hadoop，所以我们认为，Hadoop 的作用是从大量不同来源采集多种形式的数据（其中包含大量实例、事件和类），并将其转换为有价值的事物——数据产品。

第 2 章　大数据操作系统

数据团队通常是由 5 至 7 名成员组成的小型团队，他们通过敏捷方法实现由假设驱动的工作流。虽然数据科学家通常认为自己是掌握了数

据方面大量技能的通才 1，但是他们往往只精于软件、统计或领域专业知识其中之一。因此，数据团队的成员分为三大类：数据工程师负责数据的传输和原理的实践，其工作通常涉及软件和计算资源；数据分析师专注于探索和解释数据，并创建用于推断或预测的数据产品；领域专家提供过程和应用程序方面的专业知识以解决问题。

> 1Harris, Harlan, Sean Murphy, Marck Vaisman, *Analyzing the Analyzers* (http://oreil.ly/1PxPrNg)(O'Reilly, 2013).

考虑到分布式计算的技术本质，使用 Hadoop 的数据团队常常将数据科学的重中之重放在数据工程方面。相较于基于实例的方法，大数据集更适合采用基于聚合的方法，用于分布式机器学习和统计分析的大型工具集也已经存在。因此，大多数关于 Hadoop 的资料针对的是软件开发人员，他们通常精通 Java——编写 Hadoop API 所用的软件语言。此外，培训材料也倾向于关注 Hadoop 的架构方面，因为这些方面展示了 Hadoop 之所以能成功处理大型机器学习等任务的创新本源。

本书的重点是如何使用 Hadoop 进行分析，而不是如何操作 Hadoop。然而，只有对分布式计算和存储的工作原理有所了解，才能更全面地了解如何使用 Hadoop 构建用于数据处理的算法和工作流。本章将展示 Hadoop 作为大数据操作系统的一面，通过两个主要组件——分布式文件系统 HDFS（Hadoop Distributed File System）以及负载和资源管理器 YARN（Yet Another Resource Negotiator）——概述 Hadoop 的原理。本章还将演示如何使用命令行与 HDFS 交互，并执行一个 MapReduce 作业。读完本章后，你应该能够轻松地与集群进行交互，并能执行本书其余部分的示例。

2.1　基本概念

为了执行大规模计算，Hadoop 将涉及大型数据集的分析计算分发给许多机器，每台机器同时对各自的数据块进行运算。分布式计算不是新的概念，但它是一项技术挑战，需要开发分布式算法，管理集群中的机器，并实现网络和架构细节。具体来说，分布式系统必须满足以下要求。

容错性

　　如果一个组件失败，不应导致整个系统出现故障，系统应能降级到较低性能状态。如果失败的组件恢复了，它应该能够重新加入系

统。

可恢复性

发生故障时，不应丢失数据。

一致性

一个作业或任务的失败不应该影响最终的结果。

可扩展性

负载的增加（更多的数据或更多的计算）导致性能下降，而不是出现故障；资源的增加应使容量按比例增加。

Hadoop 通过以下几个抽象概念来满足上述要求。（在正确实现的情况下，这些概念定义了集群如何管理数据存储和分布式计算；此外，了解这些概念成为 Hadoop 架构基本前提的原因有助于理解其他概念，如数据流水线和分析数据流。）

- 数据添加到集群后即刻被分发出去，并存储在多个节点上。最好用节点处理本地存储的数据，以将网络流量最小化。
- 数据存储在固定大小（通常为 128MB）的块中，每个块跨系统多次复制，以提供冗余和数据安全。
- 通常将计算作为一个作业。作业被分解成任务，每个节点针对单个数据块执行任务。
- 编写作业时，不需要考虑网络编程、时间或底层的基础设施，从而允许开发人员专注于数据和计算，而不是分布式编程细节。
- 系统应该以透明的方式将节点之间的网络流量最小化。每个任务应该是独立的，并且节点也不应在处理期间彼此通信，以确保没有会导致死锁的进程间依赖。
- 作业通常通过任务冗余来容错，这使得单个节点或任务的失败不会导致最终的计算结果不正确或不完整。
- 主程序将工作分配给 worker 节点，这使得许多 worker 节点可以针对各自负责的数据进行并行运算。

虽然这些基本概念的实现在不同的 Hadoop 系统上略有不同，但它们驱动了核心架构，并确保满足容错性、可恢复性、一致性和可扩展性的要求。这些要求又确保了 Hadoop 是一个数据分析处理符合预期的数据管理系统（传统上，这些分析处理是在关系数据库或科学数据仓

库中执行的）。与数据仓库不同，Hadoop 能够在更经济的现成商业硬件上运行。因此，Hadoop 主要用于存储和计算大型异构数据集。快速分析和数据产品的原型设计有赖于这些存储在“数据湖泊”中（而不是数据仓库中）的数据集。

2.2 Hadoop架构

Hadoop 由两个主要组件组成：HDFS 和 YARN，它们实现了上一节讨论的分布式存储和计算的基本概念。HDFS（有时缩写为 DFS）是 Hadoop 的分布式文件系统，负责管理存储在集群中磁盘上的数据；YARN 则是集群资源管理器，将计算资源（worker 节点上的处理能力和内存）分配给希望执行分布式计算的应用程序。架构栈如图 2-1 所示。值得注意的是，原先的 MapReduce 应用程序和其他新的分布式计算应用程序，如图形处理引擎 Apache Giraph（http://giraph.apache.org）和内存计算平台 Apache Spark（http://spark.apache.org），现在基于 YARN 实现。

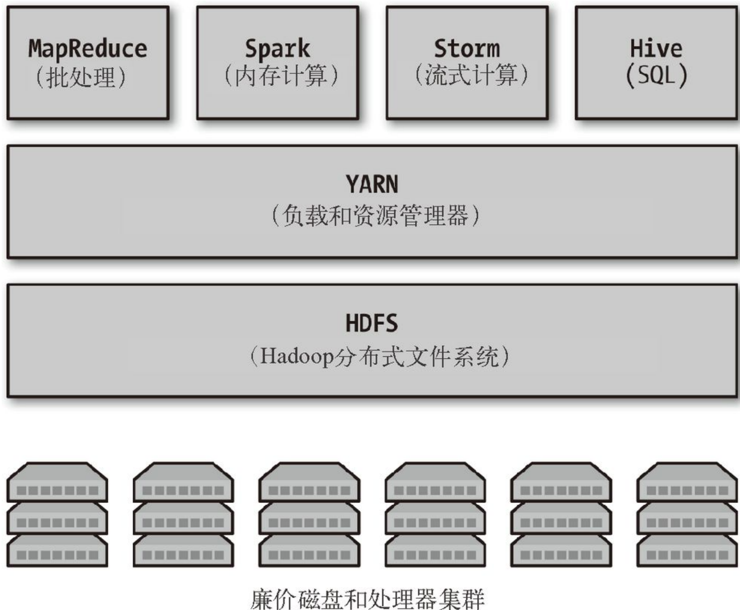


图 2-1：Hadoop 由 HDFS 和 YARN 构成

HDFS 和 YARN 协同工作，主要通过确保数据对于所需的计算是本地的，最大限度地减少集群中的网络流量。数据和任务的重复确保了容错性、可恢复性和一致性。此外，集群被集中管理，提供了可扩展

性，还可将底层的集群编程细节抽象化。HDFS 和 YARN 共同构建了大数据应用程序的平台——也许不仅仅是一个平台，它们为大数据提供了一个操作系统。

和任何优秀的操作系统一样，HDFS 和 YARN 也很灵活。除 HDFS 之外的其他数据存储系统可以集成到 Hadoop 框架中，例如 Amazon S3 或 Cassandra。此外，数据存储系统也可以直接构建在 HDFS 之上，从而提供简单文件系统之外的功能。例如，HBase 是一个构建在 HDFS 之上的列式数据存储，它是利用分布式存储的一个先进的分析应用程序。在 Hadoop 的早期版本中，如果应用程序希望利用 Hadoop 集群上的分布式计算，它就必须将用户级实现转换为 MapReduce 作业。然而，YARN 现在支持对集群功能进行更丰富的抽象，这使得用于机器学习、图形分析、类 SQL 的数据查询，甚至流式数据服务的新数据处理应用程序速度更快，且更容易实现。因此，围绕着 Hadoop，特别是 HDFS 和 YARN，丰富的工具和技术生态系统得以构建起来。

2.2.1　Hadoop集群

现在来问自己一个很有用的问题：什么是集群？到目前为止，我们一直在说 Hadoop 是一个以协调方式运行的机器集群。然而，Hadoop 并不是你必须购买或维护的硬件，它其实是运行在集群上的软件的名称，即 HDFS 和 YARN，它们由在一组计算机上运行的 6 种后台服务组成。

现在来逐一介绍它们。HDFS 和 YARN 提供了一个应用程序编程接口（Application Programming Interface，API），它使开发人员不必关注底层的集群管理细节。集群就是运行 HDFS 和 YARN 的一组计算机，每台计算机被称为一个节点。集群可以有一个节点，也可以有成千上万个节点，但是所有集群都是水平扩展的，这意味着在添加更多节点时，集群以线性方式提升容量和性能。

HDFS 和 YARN 由几个守护进程实现。守护进程是在后台运行并且不需要用户输入的软件。Hadoop 进程是服务，这意味着它们一直在集群节点上运行，接受输入并通过网络传递输出，这与 HTTP 服务器的工作原理类似。每个进程在自己的 Java 虚拟机（Java Virtual Machine，JVM）中运行，因此每个守护进程都有自己的系统资源分配，并由操作系统独立管理。集群中的每个节点都由其运行的一个或多个进程的类型标识。

master 节点

这些节点为 Hadoop 的 worker 节点提供协调服务，通常是用户访问集群的入口点。没有 master 节点，协调就不复存在，也就不可能进行分布式存储或计算。

worker 节点

集群中的大多数计算机都属于这类节点。worker 节点运行的服务从 master 节点接受任务——存储或检索数据、运行特定应用程序。worker 节点通过并行分析运行分布式计算。

HDFS 和 YARN 都有多个 master 服务，负责协调运行在各个 worker 节点上的 worker 服务。worker 节点实现 HDFS 和 YARN 的 worker 服务。HDFS 的 master 服务和 worker 服务如下所示。

NameNode（master 服务）

用于存储文件系统的目录树、文件元数据和集群中每个文件的位置。如果客户端想访问 HDFS，必须先通过从 NameNode 请求信息来查找相应的存储节点。

Secondary NameNode（master 服务）

代表 NameNode 执行内务任务并记录检查点。虽然它叫这个名字，但它并不是 NameNode 的备份。

DataNode（worker 服务）

用于存储和管理本地磁盘上的 HDFS 块，将各个数据存储的健康状况和状态报告给 NameNode。

从宏观上看，当从 HDFS 访问数据时，客户端应用程序必须先向 NameNode 发出请求，以在磁盘上定位数据。NameNode 将回复一个存储数据的 DataNode 列表，客户端必须直接从 DataNode 请求每个数据块。注意，NameNode 不存储数据，也不将数据从 DataNode 传递到客户端，而是像交警指挥交通一般，将客户端指向正确的 DataNode。

和 HDFS 类似，YARN 也有两个 master 服务和一个 worker 服务，如下所示。

ResourceManager（master 服务）

为应用程序分配和监视可用的集群资源（如内存和处理器核心这样的物理资源），处理集群上作业的调度。

ApplicationMaster（master 服务）

根据 ResourceManager 的调度，协调在集群上运行的特定应用程序。

NodeManager（worker 服务）

在单个节点上运行和管理处理任务，报告任务运行时的健康状况和状态。

与 HDFS 的工作方式类似，如果客户端希望执行作业，就必须先向 ResourceManager 请求资源，ResourceManager 会分配一个应用程序专用的 ApplicationMaster，它在作业的执行过程中会一直存在。ApplicationMaster 跟踪作业的执行，ResourceManager 则跟踪节点的状态，每个 NodeManager 创建容器并在其中执行任务。请注意，Hadoop 集群上也可能运行着其他进程（例如 JobHistory 服务器或 ZooKeeper 协调器），但这些服务是 Hadoop 集群中运行的主要软件。

主进程非常重要，所以它们通常在自己的节点上运行。因此，它们不会竞争资源，也不会带来瓶颈。但是在较小的集群中，所有的主守护进程可能都在一个节点上运行。以一个小型 Hadoop 集群的部署为例，它有六个节点，分别为两个 master 节点和四个 worker 节点，如图 2-2 所示。请注意，在较大的集群中，NameNode 和 Secondary NameNode 将驻留在不同的计算机上，从而避免竞争资源。因为集群是水平扩展的，所以集群的大小应该与预期的计算或数据存储能力呈正相关。一般来说，拥有 20～30 个 worker 节点和单个 master 节点的集群足以在几十太字节的数据集上同时运行多个作业。对于部署几百个节点的集群，每个 master 节点都拥有自己的计算机；而在拥有数千个节点的集群中，会有多个 master 节点被用于协调。

> 不一定要在集群上开发 MapReduce 作业；相反，大多数 Hadoop 开发人员通常在虚拟机中使用“伪分布式”开发环境。开发可以在一小部分数据样本上进行，而不必动用整个数据集。关于如何搭建伪分布式开发环境，请参见附录 A。

还有一种集群也很值得关注，那就是单节点集群。在“伪分布式模

式”中，单个机器将运行所有 Hadoop 守护进程，就好像它是集群的一部分，但网络流量是通过本地环回网络接口流动的。这种模式虽然没有发挥出分布式架构的优势，但却是一种完美的开发模式，因为不必为管理几台机器而费心。Hadoop 开发人员通常使用伪分布式环境，该环境通常位于虚拟机内部，通过 SSH 连接虚拟机。Cloudera、Hortonworks 和其他流行的 Hadoop 发行版提供了预先构建的虚拟机镜像，供你下载并立即使用。如果你想自己配置伪分布式节点，请参考附录 A。

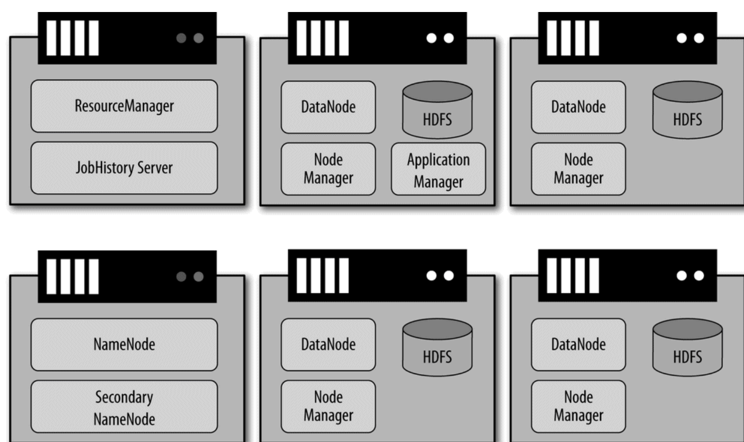


图 **2-2**：拥有两个 **master** 节点和四个 **worker** 节点的小型 **Hadoop** 集群，实现了全部六个主要的 **Hadoop** 服务

2.2.2 HDFS

通过将数据存储在由廉价、不可靠的计算机组成的集群中，HDFS 为大数据提供冗余存储，从而扩展单台计算机的可用存储容量。然而，由于分布式文件系统的网络特性，HDFS 比传统的文件系统更复杂。为了在最大程度上降低这种复杂性，HDFS 采用集中式存储架构。**2**

> 2Ghemawat、Gobioff 和 Leung 于 2003 年发表的论文“The Google File System”（http://bit.ly/google-filesystem）率先提出这一理念。

理论上，HDFS 是位于本机文件系统（如 ext4 或 xfs）之上的软件层。而实际上，Hadoop 一般化了存储层，可与本地文件系统和其他存储类型（如 Amazon S3）进行交互。HDFS 是分布式文件系统的旗舰，也是大多数编程场景所用的主要文件系统。它被设计用于存储非常大的文件，使用流访问数据，因此有一些注意事项。

- 与占用相同容量的数以亿计的小文件相比，HDFS 更适合处理数量适中但非常大的文件（例如几百万个 100MB 或更大的文件）。
- HDFS 采用 WORM 模式，即写一次，读多次（write once, read many）。它不允许随机写入或追加到文件。
- HDFS 针对文件的大型流式读取进行了优化，不采用随机读取或随机选择。

因此，用 HDFS 来存储用于计算的原始输入数据、计算阶段之间的中间结果，以及整个作业的最终结果再合适不过了。但如果应用程序需要实时更新、交互式数据分析和基于记录的事务支持，它就不适合作为数据后端使用了。相反，通过写入一次并读取多次数据，HDFS 用户可以创建大量异构数据，以进行不同的计算和分析。这些数据存储有时被称为“数据湖泊”，因为它们以可恢复和容错的方式简单地保存关于已知问题的所有数据。本书稍后将讲解突破这些限制的变通方法。

01. 文件块

    HDFS 文件分为多个块，块大小通常为 64MB 或 128MB。尽管这可在运行时配置，但是高性能系统通常将块大小设为 256MB。块大小是可以在 HDFS 中读取或写入的最小数据量，类似于单个磁盘文件系统上的块大小。但是与单个磁盘上的块不同，小于块大小的文件不占用实际文件系统上一个完整块的空间。这意味着为了实现最佳性能，Hadoop 更喜欢分解成小块的大文件，能将许多较小的文件合并成一个大文件就很好。但是如果 HDFS 上存储了许多小文件，就不是每个文件都能让总可用磁盘空间减少 128MB 了。

    块能将运行中的非常大的文件拆分并将其分发到许多计算机上。来自同一文件的不同块将被存储在不同的计算机上，以提供更高效的分布式处理。事实上，在任务和数据块之间存在一对一的关系。

    此外，块将跨 DataNode 复制。默认情况下，块将复制三份，但也可在运行时配置。因此，每个块都将分布在三台计算机和三块磁盘上。即使两个节点都发生了故障，数据也不会丢失。请注意，这意味着集群中的潜在数据存储容量仅为可用磁盘空间的三分之一，但因为磁盘通常非常便宜，所以这在大多数数据应用程序中都不成问题。

02. 数据管理

主 NameNode 记录组成文件的块和这些块所在的位置。NameNode 与 DataNode（集群中实际保存块的进程）进行通信。与每个文件相关联的元数据被存储在 NameNode 的 master 节点的内存中，以便进行快速查找。如果 NameNode 停止或发生故障，整个集群都将无法访问！

Secondary NameNode 不是 NameNode 的备份，而是代表 NameNode 执行内务任务，包括（特别是）定期将当前数据空间的快照与编辑日志合并，以确保编辑日志不会过大。编辑日志用于确保数据的一致性，防止数据丢失。如果 NameNode 发生故障，就可以用合并后的记录重建 DataNode 的状态。

当客户端应用程序想要读取文件时，它首先从 NameNode 请求元数据，以定位组成文件的块以及存储块的 DataNode 的位置。然后，应用程序直接与 DataNode 通信以读取数据。因此，NameNode 仅仅扮演着日志或查找表的角色，而不是同时读取的瓶颈。

2.2.3 YARN

虽然原始版本的 Hadoop（Hadoop 1）普及了 MapReduce，并使大众接触到了大规模的分布式处理，但它只在 HDFS 上提供 MapReduce。这是因为在 Hadoop 1 中，MapReduce 作业 / 工作负载管理功能与集群 / 资源管理功能高度耦合。因此，其他处理模型或应用程序无法将集群基础设施用于其他分布式工作负载。

虽然 MapReduce 在批量处理大规模工作负载时非常高效，但是它是 I/O 密集型的，并且由于 HDFS 和 MapReduce 的面向批处理性质，它在支持交互式分析、图形处理、机器学习和其他内存密集型算法时面临明显的限制。虽然已经为这些特定的场景开发了其他分布式处理引擎，但是 Hadoop 1 专注于 MapReduce 的本质决定了它不可能改变同一集群的用途，转而去支持这些分布式工作负载。

Hadoop 2 通过引入 YARN 突破了这些限制。YARN 将工作负载管理与资源管理分离，以便多个应用程序可以共享一个集中的公共资源管理服务。通过在 YARN 中提供通用的作业和资源管理能力，Hadoop 不再是一个仅仅专注于 MapReduce 的框架，而成为了一个完整、多应用程序的大数据操作系统。

2.3 使用分布式文件系统

请记住，HDFS 实际上是一个分布式远程文件系统。它与 POSIX 文件系统的相似性很容易误导我们，特别是文件系统查找的所有请求都发送到 NameNode。NameNode 能够快速响应查找类型的请求。一旦你开始访问文件，速度会很快慢下来，因为组成请求文件的各个块都必须通过网络传输到客户端。还要记住，因为块在 HDFS 上有多个副本，所以 HDFS 中的可用磁盘空间实际上比硬件提供的可用磁盘空间少。

> 以下示例提供的命令和环境变量可能与你使用的 Hadoop 版本或系统不同。在大多数情况下，这些差异应该很容易理解。本书假设你使用和附录 A 描述的伪分布式节点一样的设置。

在大多数情况下，与 HDFS 的交互是通过命令行接口进行的，使用过 Unix 或 Linux 上的 POSIX 接口的用户都很熟悉这种方式。此外，HDFS 还有一个 HTTP 接口和一个用 Java 编写的可编程接口。但是因为开发人员最熟悉命令行接口，所以本书将从命令行接口入手。

在本节中，我们将通过命令行完成与分布式文件系统的基本交互。假设执行这些命令的是可以连接到远程 Hadoop 集群的客户端，或本地主机上运行着伪分布式集群的客户端。此外，本书也假设 `hadoop` 命令和 $HADOOP_HOME/bin 中的其他工具位于系统路径上，并且可以被操作系统找到。

2.3.1 基本的文件系统操作

用户可以进行所有常规的文件系统操作，例如创建目录，移动、删除和复制文件，列出目录内容，修改集群上文件的权限。要查看 `fs` 命令下可用的命令，键入：

```
hostname $ hadoop fs -help
Usage: hadoop fs [generic options]
...
```

如你所见，许多与文件系统交互的常见命令都可用，通过 `hadoop fs` 命令的参数指定——就像 Java 风格的标志参数，即在命令后加短横线（-）。这些命令的辅助标志或选项另外使用由空格分隔的 Java 样式参数指定，跟在初始命令之后。请注意，指定此类选项的顺序很

重要。

开始动手吧。先使用 put 或 copyFromLocal 命令从本地文件系统复制一些数据到远程（分布式）文件系统。这两个命令是相同的，都能将文件写入分布式文件系统，而不删除本地副本。moveFromLocal 命令的功能类似，但会在文件成功传输到分布式文件系统后，将本地副本删除。

在存放本书代码和资源的 GitHub 仓库（https://github.com/bbengfort/hadoop-fundamentals）中，有一个 /data 目录，其中有一个名为 shakespeare.txt 的文件，包含了莎士比亚作品全集。将此文件下载到你的本地工作目录。下载后，将文件复制到分布式文件系统，如下所示：

```
hostname $  hadoop fs -copyFromLocal shakespeare.txt shakespeare.txt
```

本示例调用 Hadoop 的 shell 命令 copyFromLocal，并带有 <src> 和 <dst> 两个参数，它们都被指定为 shakespeare.txt 文件的相对路径。整个过程为：copyFromLocal 命令在当前工作目录中搜索 shakespeare.txt 文件，从 NameNode 请求有关该路径的信息，然后直接与 DataNode 通信以传送文件，将其复制到 HDFS 上的 /user/analyst/shakespeare.txt 路径。因为莎士比亚作品全集小于 64MB，所以它不会被分成块。但是请注意，在本地计算机以及远程 HDFS 系统上，都必须考虑相对路径和绝对路径。以上命令的全写形式如下：

```
hostname $ hadoop fs -put /home/analyst/shakespeare.txt \
           hdfs://localhost/user/analyst/shakespeare.txt
```

你可能注意到了，HDFS 上的主目录和 POSIX 系统上的主目录很像。/user/analyst/ 目录就是主目录——analyst 用户的主目录。远程文件系统的相对路径将用户的 HDFS 主目录视为当前工作目录。事实上，HDFS 文件和目录的权限模型和 POSIX 的非常像。为了更好地管理 HDFS 文件系统，像在本地文件系统上一样创建目录的分层树：

```
hostname $ hadoop fs -mkdir corpora
```

要列出远程主目录的内容，可以使用 ls 命令：

```
hostname $ hadoop fs -ls .
drwxr-xr-x   -  analyst analyst        0 2015-05-04 17:58 corpora
-rw-r--r--   3  analyst analyst 8877968 2015-05-04 17:52 shakespeare.txt
```

HDFS 文件列举命令与具备一些 HDFS 特定特征的 Unix `ls -l` 命令很像。但当此命令不带任何参数时，则会提供用户的 HDFS 主目录的列表。第 1 列显示文件的权限模式，第 2 列是文件的副本数（默认情况下，副本数为 3）。请注意，目录不会被复制，因此本例中的此列是短横线（-）。其后依次是用户、组、以字节为单位的文件大小（目录为零）、最后一次修改的日期和时间，以及文件名。

其他基本的文件操作（如 `mv`、`cp` 和 `rm`）在远程文件系统上的工作方式和预想的一样。但是，删除目录时不使用 `rmdir` 命令，而是使用 `rm -R` 递归删除目录及其包含的所有文件。

将文件从分布式文件系统读取和移动到本地文件系统时应小心处理，因为分布式文件系统维护的文件非常大。有些情况下，用户需要详细检查文件，特别是 MapReduce 作业的结果生成的输出文件。这些文件通常不被读取到标准输出流，而是使用管道传输到其他程序，如 `less` 或 `more`。

如需读取文件的内容，可使用 `cat` 命令，然后将输出通过管道传递给 `less` 以查看远程文件的内容：

```
hostname $ hadoop fs -cat shakespeare.txt | less
```

> 当使用 `less` 时，可以通过方向键导航文件，键入 `q` 退出并退回到终端。

还可以使用 `tail` 命令仅检查文件的最后 1000 字节：

```
hostname $ hadoop fs -tail shakespeare.txt | less
```

没有类似 `hadoop fs -head` 的命令可以用来检查文件的前 1000 字节。不过，可以使用 `hadoop fs -cat` 并通过管道将文件内容传输到本地 shell 的 `head` 命令。这种做法很高效，因为 `head` 命令在读取整个文件之前就终止了远程流。但是以这种方式使用 shell 的 `tail` 会降低效率，因为在计算输出之前，所有数据都必须从远程文件系统流式传输到本地文件系统。相反，`hadoop fs -tail` 命令在

远程文件中寻址到正确位置，仅通过网络返回所需的数据。

若想将整个文件从分布式文件系统传输到本地文件系统，可以使用 `get` 或 `copyToLocal`，这两个命令是相同的。与之类似，也可以使用 `moveToLocal` 命令，但它会将该文件从分布式文件系统中删除。`get merge` 命令复制符合给定模式或指定目录下的所有文件，并将其合并成本机的单个文件。如果远程系统上的文件较大，可以在管道传输的时候使用压缩工具：

```
hostname $ hadoop fs -get shakespeare.txt ./shakespeare.from-remote.txt
```

比较后可以发现，原始的 shakespeare.txt 文件与 shakespeare.from-remote.txt 文件相同。希望这些示例充分展现了 `hadoop fs` 命令是一个功能齐全的 HDFS 命令行接口，而且经常用于开发分析作业。表 2-1 展示了 `hadoop fs` 提供的其他命令，它们也都很有用。

表**2-1**：其他有用的命令

命令	输出
`hadoop fs -help <cmd>`	提供特定于 cmd 的信息和标志
`hadoop fs -test <path>`	回答各种关于 path 的问题（例如它是否存在、是否是目录、是否是文件，等等）
`hadoop fs -count <path>`	统计符合指定文件模式的路径所包含的目录数、文件数和字节数
`hadoop fs -du <path>`	显示符合指定文件模式的文件使用的空间字节数
`hadoop fs -stat <path>`	打印 path 路径的文件 / 目录的统计数据
`hadoop fs -text <path>`	获取一个源文件并以文本格式输出文件内容，目前支持 Zip、TextRecord InputStream 和 Avro 文件

2.3.2　HDFS文件权限

如前所述，HDFS 的文件权限与 POSIX 类似。权限分三种类型：读（`r`）、写（`w`）和执行（`x`）。这些权限定义了所有者、组和任何其他系统用户的访问级别。对于目录来说，执行权限允许访问目录的内容，但是 HDFS 上文件的执行权限被忽略了。在 HDFS 中，读写权限指定谁可以访问数据，以及谁可以追加文件内容。

目录列举命令 `ls` 能显示权限信息。每个模式有 10 个槽位。第 1 个槽位是 `d`，表示“是目录，否则是文件（`-`）”。接下来的槽位每 3 个为一组，分别表示所有者、组和其他用户的 rwx 权限。有几个 HDFS 的 shell 命令能管理文件和目录的权限，即我们熟悉的 `chmod`、`chgrp` 和 `chown` 命令：

```
hostname $ hadoop fs -chmod 664 shakespeare.txt
```

在上例中，chmod 命令将 shakespeare.txt 的权限更改为 -rw-rw-r--。664 是为权限三元组设置的标志的八进制表示。6 的二进制数为 110，这意味着设置了读和写的标志，但没有设置执行标志；完全允许是 7，即二进制数 111；只读是 4，即二进制数 100。chgrp 和 chown 命令分别更改分布式文件系统上文件的组和所有者。

设置 HDFS 文件权限时需要注意，客户端的身份是由跨 HDFS 运行的进程的用户名和组确定的，这意味着远程客户端可以在系统上创建任意用户。因此，这些权限只用于防止数据意外丢失，以及在已知用户之间共享文件系统资源，而不作为安全机制使用。

2.3.3 其他HDFS接口

软件开发人员可以通过 Java API 访问 HDFS 的编程接口，并且所有将数据采集到 Hadoop 集群的过程都应考虑使用该 API。还有一些可以将 HDFS 与其他文件系统或网络协议（如 FTP 或 Amazon S3）集成的工具。第 6 章将更加关注数据管理问题，以及如何从各种源获取数据并将其存储到 HDFS。

HDFS 也有 HTTP 接口，可用于集群文件系统的常规管理以及使用 Python 访问 HDFS。HDFS 的 HTTP 接口主要有两种：一是通过处理 HTTP 请求的 HDFS 守护进程直接访问，二是由代理服务器暴露 HTTP 接口，然后代表客户端使用 Java API 直接访问 HDFS。代理包括 HttpFS、Hoop 和 WebHDFS，它们都支持 RESTful 网络访问 Hadoop 集群，这很容易用 Python 实现。

通过在端口 50070 上运行的 HTTP 服务器，NameNode 也提供对 HDFS 的直接只读 HTTP 访问。如果以伪分布式模式运行，只需打开浏览器并访问 http://127.0.0.1:50070；否则，就使用集群上 NameNode 的主机名。NameNode Web UI 提供了集群状态的总览情况，包括可用和已用的存储容量、活动和死亡的 DataNode 数量，以及副本数不足的块的警告信息。

通过使用 Utilities 下拉选项卡中的搜索和导航工具，NameNode 还允许用户浏览文件系统。文件元信息的列举方式类似于命令行接口，特定文件还允许下载。通过访问 DataNode 主机的 50075 端口，可以直接浏览 DataNode 上的信息，所有活动的 DataNode 都会在 NameNode 的 HTTP 站点上列出。

默认情况下，HTTP 的直接访问接口是只读的。为了提供对 HDFS 集群的写访问，必须使用 WebHDFS 等代理。WebHDFS 通过使用 Kerberos 进行身份验证，从而保护集群。如何访问 Hadoop 上的安全资源主要取决于集群的特定配置，以及是否使用了任何第三方管理工具。Hadoop 用于运行在完全托管且不暴露于外部世界的内部集群上，因此 Hadoop 的安全性不像其平台本身那么成熟——这也是 Hadoop 继续发展需要考虑的主要问题之一。

2.4 使用分布式计算

到目前为止，你应该已经能通过命令行轻松与集群（甚至是伪分布式集群）交互了。大多数数据科学家和软件开发人员应该对上一节展示的文件系统命令很熟悉。除了在大型数据集的管理和集群中的网络通信方面有一些差异之外，HDFS 可以被很轻松地集成到当前的操作工作流中。本书接下来的内容将主要关注驻留在 HDFS 上的数据的管理和计算。为了实现这一点，我们需要对分布式计算及其要求有基本了解。

虽然 YARN 已经使 Hadoop 成为一个通用的分布式计算平台，但 MapReduce（通常缩写为 MR）是 Hadoop 的第一个计算框架。YARN 让非 MapReduce 框架（如 Spark、Tez 和 Storm，仅举几例）可以与原先的 MapReduce 应用程序一起在 Hadoop 集群上运行。但是，对于大多数 Hadoop 用户来说，MapReduce 仍然是许多应用程序和分析的主要框架。此外，对 MapReduce 的工作原理有所了解，能帮助我们更深刻地理解分布式分析，还可以讨论其他平台是如何工作的，因为 MapReduce 的理论基础与其他框架是相同的。

本节将探究 MapReduce 编程范式的基本原理，并讨论为何这些函数式编程结构会成为分布式系统的理想选择。我们将通过两个简单的分析示例（单词计数和共同好友）演示 MapReduce 如何工作，这两个示例通常用于演示分布式环境中的计算。最后，本节将描述如何在 Hadoop 集群上实现 MapReduce 应用程序，同时演示如何提交和管理示例 MapReduce 作业，并通过 Hadoop 命令行接口获取输出。

2.4.1 MapReduce：函数式编程模型

当人们提到 MapReduce 时，通常指的是分布式编程模型。3MapReduce 是一个简单但功能强大的计算框架，专门用于在集中管理的机器集群上进行容错的分布式计算。它采用了先天可并行的“函数式”编程风格来实现，允许多个独立任务在本地数据块上执行函

数，并在处理后聚合结果。

> 3分布式编程模型由 Google 设计，Jeffrey Dean 和 Sanjay Ghemawat 后来在论文“MapReduce: Simplified Data Processing on Large Clusters”（http://bit.ly/google-mapreduce-paper）中介绍了它。

函数式编程是一种编程风格，它确保每一个计算单元以无状态的方式被评估。这意味着函数仅依赖于输入，并且是封闭且不共享状态的。通过将一个函数的输出作为另一个完全独立的函数的输入，函数之间实现数据传输。这些特征使得函数式编程非常适合分布式的大数据计算系统，因为它允许我们将计算移动到任何有数据输入的节点，并保证得到的结果相同。因为函数是无状态的，且仅仅依赖于它们的输入，所以多台机器上的多个函数可以独立处理一小部分数据。通过将一些函数的输出策略性地链接到其他函数的输入，可以实现整个数据集的最终计算。

负责分发任务和聚合结果的两类函数分别被称为 map 和 reduce。这些函数运算的输入和输出数据不是简单的列表或值的集合，MapReduce 其实是利用键值对来协调计算。因此，Python 中 map 和 reduce 函数的伪代码如下所示：

```
def map(key, value):
    # 执行处理
    return (intermed_key, intermed_value)

def reduce(intermed_key, values):
    # 执行处理
    return (key, output)
```

map 函数以一系列键值对作为输入，然后在每个键值对上进行单独运算。以上伪代码按照它在 MapReduce 的 Java API 中的表示表达了这一概念：一个接受两个参数（一个键和一个值）的函数。在对输入数据执行了一些分析或变换之后，map 函数输出零个或多个键值对，在以上伪代码中表示为单个元组。整个过程将 map 函数应用于输入列表以创建新的输出列表，如图 2-3 所示。

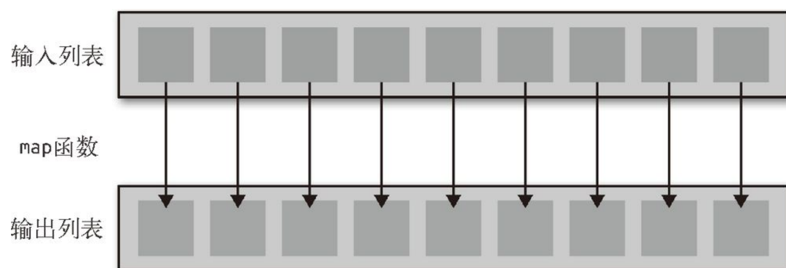


图 2-3：**map** 函数以一系列键值对作为输入，然后在每个键值对上进行单独运算，输出零个或多个键值对

通常来讲，map 操作将进行核心分析或处理，因为这个函数能查看数据集中的每个元素。想想如何在 `map` 中实现过滤器：测试每个键值对，确定它是否属于最终数据集；如果是则发出，否则就忽略。在 map 阶段之后，所有发出的键值对将按照键来分组，然后根据键被用于各个 `reduce` 函数的输入。如图 2-4 所示，`reduce` 函数被应用于一个输入列表以输出单个聚合值。

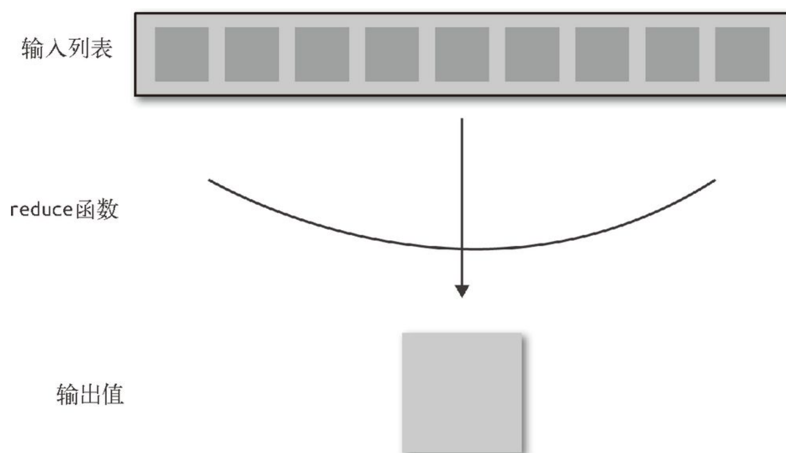


图 2-4：**reduce** 函数以一个键和一个值列表作为输入，通常通过聚合操作在整个值列表上进行运算，输出零个或多个键值对

如伪代码（与 MapReduce 的 Java API 有些类似）所示，`reduce` 函数是一个有两个参数的函数：一个键（在伪代码中是 `intermed_key`），以及与之相关联的迭代器或值列表（`values`）。reducer 对值列表执行最终处理，通常是组合或聚合，然后输出零个或多个键值对。reducer 旨在聚合从 map 阶段输出的大

量值，以便将大量数据转换为更小、更易于管理的概要数据。但它的用处可不止如此。

2.4.2 MapReduce：集群上的实现

因为 mapper 对任意列表的每个元素应用相同的函数，所以非常适合被分发到集群的节点上。每个节点获得一个 mapper 操作的副本，并且将 mapper 应用于存储在本地 HDFS 数据节点的数据块中的键值对上。独立处理数据的 mapper 的数量不定，只受集群上可用的处理器的数量限制。因为它们是无状态的，所以进程之间不需要（或不可能）进行网络通信。又因为 mapper 具有确定性，它们的输出不依赖于输入值以外的内容，所以可以在另一个节点上重试失败的 mapper。

reducer 需要根据键获取 mapper 的输出作为输入，因此 reducer 的计算也可以被分发出去。如此一来，reduce 运算的数量最多可能与 mapper 输出中的可用键一样多。可以预见，每个 reducer 都应该能看到某个独一无二的键的所有值。为了满足该要求，需要 shuffle 和 sort 操作来协调 map 和 reduce 阶段，使 reducer 的输入按键分组并排序。shuffle 和 sort 将 map 阶段的键空间分区，以便将特定键空间分配给特定 reducer。总体来说，MapReduce 的阶段如图 2-5 所示。

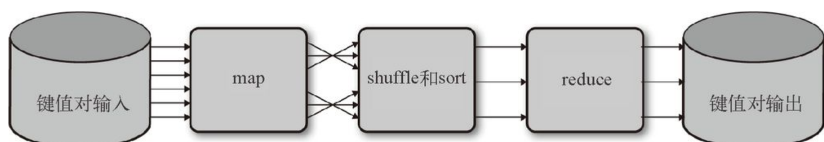


图 **2-5**：总体来说，**MapReduce** 是一个分阶段框架，其中的 **map** 阶段和 **reduce** 阶段通过中间的 **shuffle** 和 **sort** 协调

图 2-5 中涉及的阶段如下所示。

阶段 **1**

HDFS 的本地数据以键值对的形式被加载到一个映射过程。

阶段 **2**

mapper 输出零个或多个键值对，将计算所得的值映射到一个特定的键上。

阶段 **3**

基于键对这些键值对进行 sort 和 shuffle 操作，然后将它们传递给 reducer，使 reducer 获得键的所有值。

阶段 **4**

reducer 输出零个或多个最终的键值对，即输出（归约 map 的结果）。

在大多数情况下，数据工程师只需要关注 MapReduce 的这一宽泛描述便可以实现分析应用程序。但在集群上执行 MapReduce 时，还需要其他详细信息。例如，键值对是如何定义的？对键空间进行正确分区需要什么？你也可能需要进行改进和优化，如加入 combiner 和其他中间阶段，从而使简单的作业仅通过较少的计算资源就能完成。在拥有几个节点的集群上，MapReduce 流水线的数据流细节（尽管这超出了本书的范畴）如图 2-6 所示。

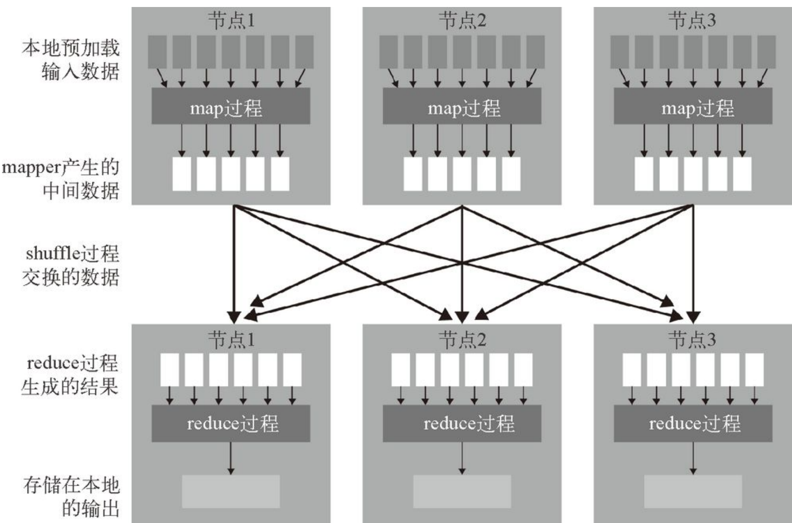


图 **2-6**：在拥有几个节点的集群上执行的 **MapReduce** 作业的数据流

在集群执行上下文中，map 任务被分配给集群中的一个或多个节点，这些节点包含指定为 map 操作输入的本地数据块。块存储在 HDFS 中，并由 `InputFormat` 类拆分为更小的块，该类定义了如何将数据呈现给 map。例如，给定文本数据，键可以是文件标识符和行号，值可以是行内容的字符串。`RecordReader` 将每个键值对呈现给用户提供的 map 操作，map 再输出一个或多个中间键值对。这时一般会使用 combiner 进行优化，它聚合单个 mapper 的输出，和 reducer

的工作原理类似，但是不涉及整个键空间。这项预备工作减少了 reducer 的工作量，能提高其性能。

中间阶段的键从 map 过程被拉到 partitioner，partitioner 决定如何将键分配给 reducer。通常假定键空间是均匀分布的，因此散列函数用于在 reducer 之间均匀地划分键。partitioner 还对键值对进行排序，以便实现完整的“shuffle 和 sort”阶段。最后，reducer 开始工作，为每个键生成数据迭代器并执行 reduce 操作，如聚合。输出的键值对将使用 `OutputFormat` 类写回到 HDFS。

在 MapReduce 集群执行上下文中，也有其他许多管理大规模作业的工具。仅举几个例子，`Counter` 和 `Reporter` 对象用于作业跟踪和评估，缓存用于在处理期间提供辅助数据。高级框架（如 Pig 或 Hive）通常都实现了这些工具，供开发人员使用。但在第 3 章中，我们将看到如何使用 Python 和 Hadoop Streaming 来实现这些功能。

MapReduce示例

为了演示数据流经 map 和 reduce 的过程，我们将演示两个具体的例子：单词计数和共同好友。这两个应用程序虽然简单，但却能演示分布式系统内的数据流动过程。特别是单词计数，因为它经常被用于演示分布式计算任务，所以通常被称为大数据界的“Hello, World”。因为单词计数和共同好友属于“易并行”问题，所以它们不仅能帮助我们理解 MapReduce，还能指出应用程序的设计是否存在根本缺陷。

单词计数应用程序以一个或多个文本文件作为输入，生成一份单词及其频率的列表。具体来说，因为 Hadoop 使用键值对，所以输入的键是文件 ID 和行号，输入的值是字符串，而输出的键是单词，输出的值是一个整数。我们立刻就会发现，这可以采用多种方式并行化。首先，每个 mapper 可以处理单个文档；如果文档非常大，mapper 可以处理单个文档中的多个块——map 操作不关心单词的上下文，它只统计作为输入的单词的个数。同理，可以用多个 reducer 同时处理不同的键，因为输出键是一个单词。以下 Python 伪代码展示了如何实现此算法：

```
# emit是一个执行Hadoop I/O的函数 ❶

def map(dockey, line):
    for word in Line.split():
        emit(word, 1)

def reduce(word, values):
    count = sum(value for value in values)
```

```
    emit(word, count)
```

❶ 从参数的角度考虑，emit 是一个执行 Hadoop I/O 的函数；也就是说，它将其参数发送到 MapReduce 流水线的下一个阶段，类似于 Python 中的 yield 函数。

在图 2-7 中有两个文档，包含两个简单句子。map 函数将接收文本的某个唯一 ID，以及该文档内容的字符串。它通过空格和标点符号分割值（获取所有单词），并将每个单词作为中间键、值为 1 发出——因为 mapper 已经看到该单词出现了一次。每个 mapper 的数据如下所示：

```
# WordCount mapper的输入

(27183, "The fast cat wears no hat.")
(31416, "The cat in the hat ran fast.")

# mapper 1的输出

("The", 1), ("fast", 1), ("cat", 1), ("wears", 1),
("no", 1), ("hat", 1),(".", 1)

# mapper 2的输出

("The", 1), ("cat", 1), ("in", 1), ("the", 1),
("hat", 1), ("ran", 1),("fast", 1),(".", 1)
```

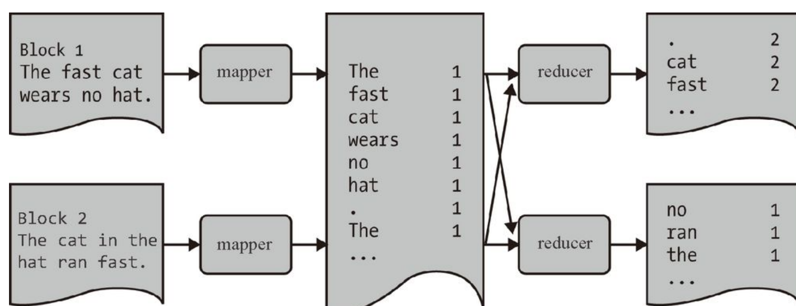


图 **2-7**：在集群上执行的有两个 **mapper** 和两个 **reducer** 的单词计数作业的数据流

这些数据被传递到 shuffle 阶段和 sort 阶段，键（单词）被分组并排序，然后发送到适当的 reducer。每个 reducer 接收以单词作为键、一串数字 1 作为值的输入。为了获得计数，它简单地将这些数字 1 相

加，并将单词作为键、计数作为值发出。示例中的输入和输出的数据如下所示：

```
# WordCount reducer的输入
# 该数据由shuffle和sort计算

(".", [1, 1])
("cat", [1, 1])
("fast", [1, 1])
("hat", [1, 1])
("in", [1])
("no", [1])
("ran", [1])
("the", [1])
("wears", [1])
("The", [1, 1])

# 所有WordCount reducer的输出

(".", 2)
("cat", 2)
("fast", 2)
("hat", 2)
("in", 1)
("no", 1)
("ran", 1)
("the", 1)
("wears", 1)
("The", 2)
```

这种算法看似简单，但是它稍微复杂一点的实现常被用于文本处理。想象一下如何计算《纽约时报》或 Google 图书语料库中最常出现的单词——这肯定需要某种大数据技术。使用 *n*-gram 语言模型可以对同时出现的单词进行计数，以查看一起出现的两个单词之间是否存在统计意义，如 white house（白宫）或 baseball bat（棒球棒）。此外，了解数据如何从输入源通过 map 操作流到 reduce 操作再产生输出，对于在分布式环境中开发分析过程和数据工程任务至关重要。

接下来学习一个稍微复杂些的例子，以确保 MapReduce 有意义。共同好友任务的目标是通过分析社交网络，查看用户间有哪些共同好友。这既是进行下游分析（例如"你可能认识的人"推荐）的第一步，也是实现只允许你与朋友及朋友的朋友分享内容的社交网络的一个关键部分。给定输入数据源，其中键是用户的名称，值是用逗号分隔的朋友列表，以下 Python 伪代码演示如何执行此计算：

```
def map(person, friends):
```

```
    for friend in friends.split(","):
        pair = sort([person, friend])
        emit(pair, friends)

def reduce(pair, friends):
    shared = set(friends[0])
    shared = shared.intersection(friends[1])
    emit(pair, shared)
```

mapper 从初始数据集创建中间键空间，其中包含所有可能存在的 `(friend, friend)` 元组。因为值是好友列表，所以可以针对每个关系分析数据集。此外，请注意该关系对已经被排序，这确保了 `("Mike", "Linda")` 和 `("Linda", "Mike")` 在 reducer 聚合时是相同的键。输入和 mapper 输出如下所示：

```
# 输入（键 → 值）
Allen → Betty, Chris, David
Betty → Allen, Chris, David, Ellen
Chris → Allen, Betty, David, Ellen
David → Allen, Betty, Chris, Ellen
Ellen → Betty, Chris, David

# mapper 1的输出
(Allen, Betty) → (Betty, Chris, David)
(Allen, Chris) → (Betty, Chris, David)
(Allen, David) → (Betty, Chris, David)

# mapper 2的输出
(Allen, Betty) → (Allen, Chris, David, Ellen)
(Betty, Chris) → (Allen, Chris, David, Ellen)
(Betty, David) → (Allen, Chris, David, Ellen)
(Betty, Ellen) → (Allen, Chris, David, Ellen)

# mapper 3的输出
(Allen, David) → (Allen, Chris, David, Ellen)
(Betty, David) → (Allen, Chris, David, Ellen)
(Chris, David) → (Allen, Chris, David, Ellen)
(David, Ellen) → (Allen, Chris, David, Ellen)

# mapper 4的输出
(Betty, Ellen) → (Betty, Chris, David)
(Chris, Ellen) → (Betty, Chris, David)
(David, Ellen) → (Betty, Chris, David)
```

针对数据集中存在的每对朋友关系，reducer 确定可以看到两个好友列表，每个好友列表对应键中的一个用户。因此，为了执行最终聚合，reducer 先简单地将这些列表转换成集合，并求两者间的交集，

即共同好友；然后，发射该交集，其中包含按字母顺序排列的关系元组和相关的好友。请注意，reducer 可以简单地为关系中的每个人发射结果，这可能对其他应用程序的下游数据加载有帮助。流入 reducer 的数据如下所示：

```
# 经过shuffle和sort之后，reducer的输入

(Allen, Betty) → (A C D E) (B C D)
(Allen, Chris) → (A B D E) (B C D)
(Allen, David) → (A B C E) (B C D)
(Betty, Chris) → (A B D E) (A C D E)
(Betty, David) → (A B C E) (A C D E)
(Betty, Ellen) → (A C D E) (B C D)
(Chris, David) → (A B C E) (A B D E)
(Chris, Ellen) → (A B D E) (B C D)
(David, Ellen) → (A B C E) (B C D)

# reduce之后

(Allen, Betty) → (Chris, David)
(Allen, Chris) → (Betty, David)
(Allen, David) → (Betty, Chris)
(Betty, Chris) → (Allen, David, Ellen)
(Betty, David) → (Allen, Chris, Ellen)
(Betty, Ellen) → (Chris, David)
(Chris, David) → (Allen, Betty, Ellen)
(Chris, Ellen) → (Betty, David)
(David, Ellen) → (Betty, Chris)
```

本节中的具体示例（单词计数和共同好友）展示了数据流经 MapReduce 作业的过程，并给出了如何开发这种作业的思路——从想象 map 阶段和 reduce 阶段的数据流动过程开始就不错。确定需要输入和输出的键也有助于指导流水线的每个阶段应该做什么。

2.4.3　不止一个**MapReduce**：作业链

将常规解决问题的工作流稍作转变以满足 `map` 函数和 `reduce` 函数的无状态运算和交互后，就可以用 MapReduce 轻松实现许多算法或数据处理任务。但是单个 MapReduce 作业无法实现更复杂的算法和分析。例如，许多机器学习或预测分析技术需要优化，即最小化误差的迭代过程。MapReduce 不支持通过单个 map 或 reduce 进行迭代。

看来有必要对这些术语进行一番讨论。在 MapReduce 中，作业实际上指的是完整的应用程序（application 或 program），即对所有输入

数据执行 `map` 函数和 `reduce` 函数的完整过程。复杂的分析作业通常由许多内部任务组成，其中的任务是指对一个数据块执行单个 `map` 运算或 `reduce` 运算的过程。因为有许多 worker 节点在同时执行类似的任务，所以一些数据处理工作流可以运行“只有 map”或“只有 reduce”的作业。例如，分箱方法可以利用内置的 partitioner 将类似的数据分在一组。分箱后的数据可以用于下游的其他 MapReduce 作业，执行频率分析或计算概率分布。

事实上，更复杂的应用程序是通过被称为“作业链”的过程，使用多个 MapReduce 作业执行单个计算来构建的。如图 2-8 所示，通过创建流经中间 MapReduce 作业系统的数据流，可以创建一个分析步骤的流水线，引导我们得到最终结果。分析人员和开发人员的工作是设计实现 `map` 和 `reduce` 的算法，以得到单一的分析结论，这部分将在第 3 章进行详细探讨。

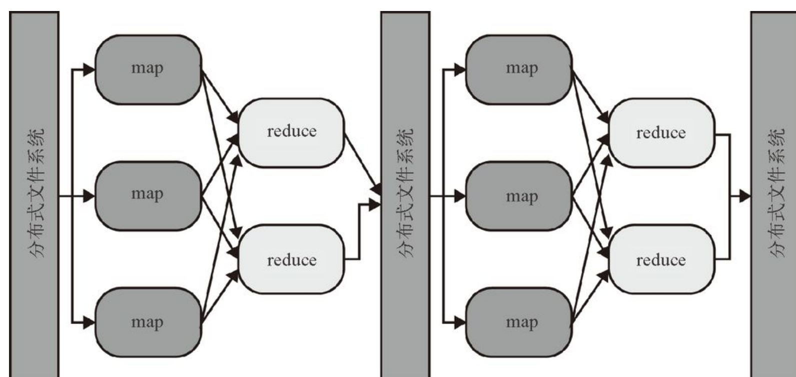


图 **2-8**：复杂的算法或应用程序实际上是通过 **MapReduce** 作业链接而成的，其中下游 **MapReduce** 作业的输入是最近的上游作业的输出

本书探讨如何将计算框架从传统的迭代分析转变为可用于大规模计算的“数据流”。数据流是作业或运算的有向无环图，被用于针对大型数据集实现某种最终计算。最终，大数据应用程序的主要数据工程工作是过滤和聚合大型数据集，以进行最后一英里计算——数据可以放入内存进行计算。显而易见，链式作业适合这种数据处理模型，其他数据处理系统（如 Storm 和 Spark）也与它有关。

2.5　向YARN提交MapReduce作业

MapReduce 的 API 是用 Java 编写的，因此提交给集群的 MapReduce 作业是编译好的 Java 归档（Java Archive，JAR）文

件。Hadoop 将 JAR 文件通过网络传输到运行任务（mapper 或 reducer）的每个节点，并执行 MapReduce 作业的各个任务。

> 虽然本书探索了几种编写 Hadoop 分析作业的方法，但是我们的应用程序将主要通过 Python 编写，使用 MapReduce Streaming 或 Spark。在某些情况下，本书还将使用 Hive 和 Pig 演示在集群上执行数据分析的其他方法。

单词计数示例演示了分布式计算的强大功能，以及如何计算非结构化数据。请从 Hadoop Fundamentals 仓库（https://github.com/bbengfort/hadoop-fundamentals）下载单词计数的 Java 示例程序 WordCount.zip，其中包含以下文件。

WordCount.java

　　执行作业的 MapReduce 驱动类。

WordMapper.java

　　发射单词的 `mapper` 类。

SumReducer.java

　　统计单词的 `reducer` 类。

使用如下命令将 Hadoop 作业编译成一个 JAR 文件：

```
hostname $ hadoop com.sun.tools.javac.Main WordCount.java
hostname $ jar cf wc.jar WordCount*.class
```

这会在当前工作目录中创建一个 wc.jar 文件。请注意，这假定几个环境变量已被正确配置，包括 `JAVA_HOME` 和 `HADOOP_CLASSPATH`。有关环境变量的详细信息，请参见附录 A。

为了将作业提交到集群并计算莎士比亚作品全集的字数，使用 `hadoop jar` 命令。该命令连接到 ResourceManager，并发送 wc.jar 文件，使其在集群的所有节点上执行。该命令需要作业归档文件的路径，以及要调用的 `main` 方法所在的类名。然后，将其他命令行参数传递到作业本身。这个简单程序需要待分析数据的输入路径，以及写入结果的输出路径。输入路径和输出路径都是 HDFS 路径，并

且输出路径不能在分布式文件系统上，否则会出现错误（为了防止覆盖或删除集群上的数据）。作业提交如下所示：

```
hostname $ hadoop jar wc.jar WordCount shakespeare.txt wordcounts
```

作业将执行并输出 mapper 和 reducer 的状态，并且在完成时报告作业完成情况的统计信息。一旦完成，作业的结果将写入 wordcounts 目录，可以按如下方式查看该目录：

```
hostname $ hadoop fs -ls wordcounts
```

可以看到几个名字类似于 part-00000 的输出文件。事实上，在计算中使用的每个 reducer 都应该有一个 part 文件。此外，还应该有一个 _SUCCESS 文件和一个 _logs 目录，用于存储有关作业的信息。为了读取作业的结果，针对远程文件系统中的 part 文件执行 `cat` 命令，并通过管道传输给 `less`：

```
hostname $ hadoop fs -cat wordcounts/part-00000 | less
```

如果 MapReduce 作业出现问题，你得能停止它。（试想一下，你不小心向 mapper 或 reducer 添加了无限循环或内存密集型进程！）但是键入 Ctrl + C（在 Unix 上发出键盘中断）只能终止显示进度的进程，而不能真正停止作业！ `hadoop job` 命令让你能管理当前运行在集群上的作业。使用 `-list` 命令列出所有正在运行的作业：

```
hostname $ hadoop job -list
```

通过输出来标识要终止的作业的 ID，然后通过 `-kill` 命令终止该作业：

```
hostname $ hadoop job -kill $JOBID
```

与 NameNode 的 Web 接口类似，ResourceManager 也提供了一个 Web 接口来查看作业的状态及日志文件。可以通过托管 ResourceManager 服务的机器的 8088 端口访问 ResourceManager 的 Web UI，此 Web UI 显示所有当前正在运行的作业，以及集群中

NodeManager 的状态。ResourceManager 不跟踪作业的历史记录，但是可以通过 JobHistory 服务器访问历史记录——在托管 JobHistory 服务器的机器的 19888 端口上访问 JobHistory 服务器。

2.6 小结

本章介绍了关于 Hadoop 集群架构的大量细节，并简要介绍了大规模分布式计算系统的需求和实现的许多要点。然而，这并不意味着我们已经介绍了全部内容——相反，目前只是为介绍本书中的概念提供了足够的背景知识。之所以要如此介绍 MapReduce 的概念细节，是为了给开发分布式算法打下基础。在此基础上，本书将继续讨论更复杂的分析算法，让你了解它们的工作原理。不过，本书不会对具体的实现进行更深入的讨论。

因为本书的目标是成为 Hadoop 分布式计算的入门指导，所以它不会关注 Hadoop 集群的搭建、配置或维护，而是关注分析人员与 Hadoop 的交互。因此，下一章将通过 Hadoop Streaming，研究如何使用 Python 编写简单的 MapReduce 分布式作业。

第 3 章　Python 框架和 Hadoop Streaming

当前版本的 Hadoop MapReduce 是一个软件框架，用于编写在集群上并行处理大量数据的作业，也是 Hadoop 自带的原生分布式处理框架。该框架提供一个 Java API，允许开发人员指定 HDFS 上的输入输出位置、`map` 和 `reduce` 函数以及其他作业参数（比如作业配置）。作业被编译并打包成 JAR 文件，作业客户端将其提交给 `ResourceManager`，这一步通常通过命令行完成。然后，`ResourceManager` 调度任务，监控任务，并将状态提供给客户端。

通常，MapReduce 应用程序由 3 个 Java 类组成：`Job`、`Mapper` 和 `Reducer`。mapper 和 reducer 处理键值对计算的细节，通过 shuffle 阶段和 sort 阶段连接。作业通过指定要从 HDFS 序列化的数据的 `InputFormat` 和 `OutputFormat` 类，来配置输入数据和输出数据的格式。所有这些类都必须扩展抽象基类或实现 MapReduce 中的接口。毋庸多言，开发 Java MapReduce 应用程序是非常复杂的。

但是，Java 不是 MapReduce 框架的唯一选择。例如，C++ 开发人员可以使用 **Hadoop Pipes**，它提供了一个能使用 HDFS 和 MapReduce 的 API。但数据科学家最感兴趣的还是 **Hadoop Streaming**，这是一个用 Java 编写的实用程序，可以将任何可执行程序指定为 mapper 或 reducer。通过 Hadoop Streaming，shell 实用程序、R 或 Python、脚本都可以用于编写 MapReduce 作业，这使数据科学家可以轻松地将 MapReduce 集成到他们的工作流中，特别是在不需要大量软件开发的日常数据管理任务中。

> Hadoop Streaming 看上去似乎不是 Hadoop 生态系统第一梯队的成员。事实上，大多数 Hadoop 用户在直接使用 Hadoop MapReduce 之前，很可能使用过更高级别的工具，例如 Pig 和 Hive。虽然 Streaming 社区的规模很小，但是有很多框架都是基于它构建的，而且有许多云计算 MapReduce 资源（如 Amazon 的 Elastic MapReduce）原生包含 Streaming。敏捷数据科学利用脚本语言和 Hadoop Streaming 的快速开发，可以快速构建数据分析乃至大规模计算作业，包括机器学习任务。

本章将探讨使用 Hadoop Streaming 的细节，并创建一个小框架，从而使用 Python 快速编写 MapReduce 作业。本章将扩展在第 2 章中使用的简单 WordCount 程序，以便使用 Python 的第三方库进行自然语言处理（natural language processing，NLP）；此外，还会编写一个用于识别文本中重要短语（bigram）频率的 MapReduce 作业；最后，将讨论一些高级的 MapReduce 主题，这对于如何理解 Hadoop，以及如何将这些主题应用于 Python 编写的 Streaming 作业中至关重要。

3.1 Hadoop Streaming

Hadoop Streaming 是一个实用程序，被打包为 Hadoop MapReduce 发行版附带的 JAR 文件。Streaming 作业像普通 Hadoop 作业一样，通过作业客户端传递到集群。但除了可以指定输入和输出的 HDFS 路径的参数外，它还可以指定 mapper 和 reducer 的可执行程序。然后，作业作为普通 MapReduce 作业运行，依然由 `ResourceManager` 和 `MRAppMaster` 管理和监控，直到作业完成。

为了执行 MapReduce 作业，Streaming 利用标准 Unix 流进行输入和

输出，因此得名 **Streaming**。mapper 和 reducer 的输入都是从 `stdin` 读取的，Python 进程可以通过 `sys` 模块访问 `stdin`。Hadoop 要求由 Python 编写的 mapper 和 reducer 将它们输出的键值对写到 `stdout` 中。图 3-1 演示了 MapReduce 中的这个过程。虽然使用 Python 的 Hadoop 开发人员不一定能够通过这种技术访问完整的 MapReduce API（partitioner、输入和输出格式等功能必须用 Java 编写），但这已足以实现数据科学家工作流中许多功能强大的常见作业和任务了。

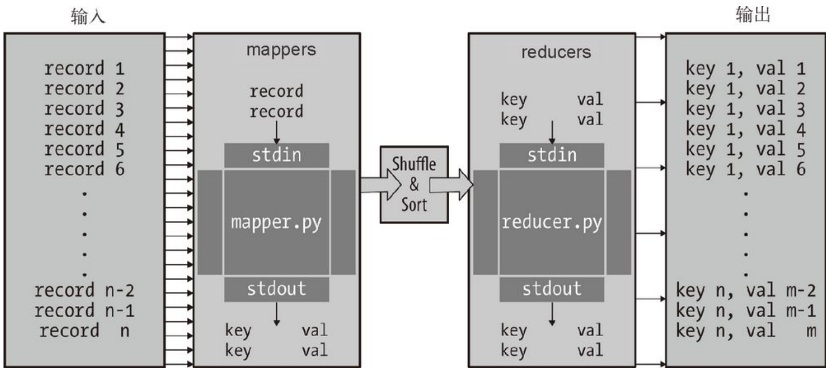


图 **3-1**：使用 **Python** 编写的 **mapper.py** 和 **reducer.py** 的 **Hadoop Streaming** 中的数据流

> 不要把 Hadoop Streaming 与 Spark Streaming 或其他使用“无界数据流”的实时计算框架（如 Apache Storm）弄混了。Hadoop Streaming 中的“流”指的是标准的 Unix 流 `stdin`、`stdout` 和 `stderr`，而 Spark Streaming 和 Storm 对一定时间内从窗口流入的数据进行实时分析——它们截然不同！本章所说的“Streaming”具体是指 Hadoop Streaming。

当 Streaming 执行作业时，每个 mapper 任务将在自己的进程内启动提供的可执行文件；然后，将输入数据转换为文本行并将其输送到外部进程的 `stdin` 的同时，从 `stdout` 收集输出。输入数据的转换通常是直接将值序列化，因为数据是从 HDFS 读取的，其中每行都是一个新值。mapper 要求输出是键或值格式的字符串，其中键和值通过某个分隔符分隔，默认为制表符（`\t`）。如果没有分隔符，mapper 就认为输出只有键，值为 `null`。可以通过向 Hadoop Streaming 作业传递参数来定制分隔符。

对 mapper 的输出进行 shuffle 和 sort 之后（确保每个相同的键都发送给同一个 reducer），reducer 也启动了可执行文件。mapper 输出的键值字符串通过 `stdin` 传输到 reducer 作为输入，reducer 的输入和 mapper 的输出相互匹配，并保证按键分组。reducer 发送到 `stdout` 的输出的格式应该与 mapper 的键、分隔符和值的格式相同。

因此，为了使用 Python 编写 Hadoop 作业，需要创建两个 Python 文件：mapper.py 和 reducer.py。只需要在这两个文件中导入 `sys` 模块，就可以访问 `stdin` 和 `stdout`。代码本身需要以字符串的形式处理输入、解析和转换每个数字或复杂的数据类型，我们也需要将输出序列化为字符串。为了演示它是如何工作的，我们将尽可能使用简单的 Python 方法来实现第 2 章讨论的 WordCount 示例。

首先，创建可执行 mapper 文件 mapper.py：

```
#!/usr/bin/env python

import sys

if __name__ == "__main__":
    for line in sys.stdin:
        for word in line.split():
            sys.stdout.write("{}\t1\n".format(word))
```

mapper 只是简单地从 `sys.stdin` 读取每一行，使用空格拆分该行文本，然后逐行将得到的每个单词和数字 1 写入 `sys.stdout`，并用制表符将两者分隔。reducer 则更复杂一些，因为针对每行输入，我们都要记录正在处理哪个键，只有看到一个新键时，才能发射一个完整的和。这是因为与本地 API 不同，单个数据值会在 shuffle 和 sort 期间聚合到流进程中，而不是暴露为一个列表或迭代器。请记住，每个 reducer 任务都可以看到同一个键的所有值，但也可以看到多个键。在 reducer.py 文件中实现的 reducer 如下所示：

```
#!/usr/bin/env python

import sys

if __name__ == '__main__':
    curkey = None
    total = 0
    for line in sys.stdin:
        key, val = line.split("\t")
        val = int(val)
```

```
        if key == curkey:
            total += val
        else:


        if curkey is not None:
            sys.stdout.write("{}\t{}\n".format(curkey, total))

    curkey = key
    total = val
```

当 reducer 迭代 `stdin` 输入中的每一行时，它会根据分隔符拆分该行并将值转换为整数。然后，它执行检查以确保仍在为同一个键计数；如果不是同一个键，则将输出写入 `stdout` 并重新启动新键的计数。mapper 和 reducer 都在“ifmain”块中执行，本章后面会讨论这部分内容。

> 如果你学习过如何使用 Java 进行 MapReduce 编程，你可能会以为 `stdin` 和 MapReduce 的 API 一样，一次只接收一行记录。但是通过 Streaming，mapper 可以访问块中的每一行，并将整个数据集视为单个项目。此外，reducer 也不像在 Java API 中那样接收累积值，而是接收从 mapper 输出的、经过排序的逐行输入。我们将使用 `groupby` 来模拟积累过程，但它不是原生的。

每个 Python 模块都在自己的进程内执行，因此它拥有运行时所有可用的处理和内存资源。但需要注意的是，由于 Hadoop Streaming 把每个 mapper 和 reducer 都视为可执行程序，所以每个 Python 文件都应以 `#!/usr/bin/env python` 开头，从而提示 shell 应该使用 Python，而不是 bash 来解释代码。

既然我们已经充分了解了 Hadoop Streaming 的工作原理，那么就来挑战一下更复杂的代码，想一想可被不同 Streaming 作业重用的高质量 Python 代码，并具体研究如何使用 Hadoop Streaming 来解析 CSV 数据。

3.1.1　使用Streaming在CSV数据上运行计算

虽然 Python 脚本在 Hadoop Streaming 中要做的全部工作只是读取 `stdin` 的内容并将其写入到 `stdout`，但是我们仍然可以改进一下

前面的代码，比如可以使用 Python 标准库中的模块进行快速迭代、字符串处理等。本节将开始搭建一个小型的可重用框架，从而为大数据处理需求快速部署 Hadoop 作业。在开始之前，先来看一个读取 CSV 数据的特定示例。

因为 mapper 和 reducer 的输入和输出是字符串，所以我们得仔细思考应该在系统中使用什么数据类型，以及希望 Python 脚本做多少解析工作。例如，可以使用内置的 `ast.literal_eval` 来解析简单数据类型（例如数字、元组、列表、字典或布尔），或者用结构化的序列化格式（例如 JSON 甚至 XML）来输入和输出复杂数据结构。因为 Streaming 逐行进行序列化，所以 Python Streaming 作业非常适用于处理 CSV 文件和其他纯文本文件，在我们的数据集和其他半结构化数据存储中就经常见到这些格式。稍后将讨论其他类型，如 Avro 或其他可以使用的二进制序列化格式。

在这个例子中，我们将使用美国国内航班准点情况数据集。该数据集由美国交通部交通统计局提供，可以从其网站（http://bit.ly/rita-transtats）下载。（本书的 GitHub 仓库中也有一个该数据集整理后的版本。）美国交通统计局提供了一个 CSV 文件，其中包含每个美国国内航班及其相关运输统计数据，如到达或离港延误，可用于分析。在本例中，整理过后的数据集每行包含的 CSV 数据有：航班日期、航空公司 ID、航班号、始发机场和到达机场、起飞时间和延误分钟、到达时间和延误分钟，最后是空中飞行时间以及里程。

```
2014-04-01,19805,1,JFK,LAX,0854,-6.00,1217,2.00,355.00,2475.00
2014-04-01,19805,2,LAX,JFK,0944,14.00,1736,-29.00,269.00,2475.00
```

通过计算每个机场的平均起飞延误时间，我们将示范怎样编写结构化的 MapReduce Python 代码。先来看看 mapper，在 mapper.py 文件中写入如下代码：

```
#!/usr/bin/env python

import sys
import csv

SEP = "\t"


class Mapper(object):

    def __init__(self, stream, sep=SEP):
        self.stream = stream
```

```
        self.sep    = sep

    def emit(self, key, value):
        sys.stdout.write("{}{}{}\n".format(key, self.sep, value))

    def map(self):
        for row in self:
            self.emit(row[3], row[6])

    def __iter__(self):
        reader = csv.reader(self.stream)
        for row in reader:
            yield row


if __name__ == '__main__':
    mapper = Mapper(sys.stdin)
    mapper.map()
```

来逐行看一下这段代码。第一行的 `#!`（声明 shebang）告诉 Linux（具体来说是 bash）使用什么程序来执行这个脚本——本例使用默认环境中的 Python，不管它是什么版本。这么简单的一行代码就创建了可执行脚本，并让 Hadoop Streaming 知道如何处理我们的文件。

后面的几行代码导入了 Python 标准库中的两个模块——`sys`（用于访问 `stdin` 和 `stdout`）以及 `csv`（用于快速解析 CSV 数据）。请注意，因为这些模块都在标准库中，所以在集群中的每个节点上都可以使用它们。第三方包和自定义代码则必须进行特殊处理，之后的内容将讨论这个问题。

我们没有创建一个过程式的脚本处理输入，而是将所有代码都写在 `Mapper` 类中。虽然 Python 实现了函数式编程技术，但它也是一种完全面向对象（object-oriented，OO）的编程语言。因为 Python 是一种解释型语言，所以它的用途广泛，从用于系统管理的便捷脚本到使用 OO 设计的大规模软件库和代码库，都能创建松耦合系统。在示例代码中，我们使用类创建了一个可扩展的 API，可以将其用于我们所有的 MapReduce 任务。这段代码的目标是可重用和可用于生产。在 3.2 节中，我们将把在此示例中学到的内容结合起来，构建一个完整的微框架，用于部署使用 Python 的 Hadoop Streaming 代码。

航班平均延误时间示例的 `Mapper` 类的实例化需要 `infile` 和 `separator` 这两个参数，它们都有默认值。`infile` 指向接收数据的位置，默认情况下为 `stdin`，这是 Hadoop 的预期值。但你可以

将这段代码修改成一个能够分析独立文件的通用框架，让它变得 DRY（don't repeat yourself，不做重复的事情），从而进行各种规模的分析。Hadoop 使用键值对进行计算，因此第二个参数用于确定输入／输出字符串的哪个部分是键，哪个部分是值。默认情况下，分隔符是制表符（`\t`）——它是一个模块级“常量”，允许我们在需要时快速重新定义分隔符。

下一个要注意的是在 `mapper` 类上使用的 `__iter__` 内置方法。双下划线通常表示这是 Python 中的一个特殊方法或函数。具体来说，`__iter__` 函数的实现使该类成为可迭代的，该函数返回一个生成器（一般通过 `yield` 语句构造），这是另一个可迭代的对象；该函数如果简单地返回 `self`，就必须同时实现 `next` 或 `__next__` 方法，这两个方法在迭代完成时抛出 `StopIteration`。这个类现在可以用于 `for` 语句，如：

```
for item in Mapper():
    print item
```

执行这行代码时，Python 会调用 `__iter__` 方法来确定如何迭代 `mapper` 类的实例——本例通过使用 `csv.reader` 解析 `stdin` 的每一行，并产生每一行。我们的类在 `map` 方法中将自己作为迭代器，因此可以循环遍历 `self` 中的每一行，也就简单地循环遍历了 `infile` 中的每一行。然后，它将输出始发机场（位置 3）并将此作为键，将起飞延误时间（位置 6）作为值，再使用 `emit` 方法发射出去——简单地将由 `sep` 分隔的键和值作为单行写入 `stdout`。

这段代码的最后一部分是 `if __name__ =="__main__"` 块，也被称为“ifmain”。在 Python 中，只有脚本作为程序的主入口点运行时，此条件才会被触发，被导入的脚本不会运行该方法。Python 开发人员使用它来判断代码是否在一个库中，或者确保任何执行的代码都在 Python 脚本的底层运行，以方便调试。通过这个语句，可以确保这个块只有在作为 mapper 直接传递给 Hadoop Streaming 时才被执行；如果是导入代码以继承 mapper（例如我们的微框架），这个块就不会被执行。现在来看看 reducer，在 reducer.py 文件中写入如下代码：

```
#!/usr/bin/env python

import sys

from itertools import groupby
```

```
from operator import itemgetter

SEP = "\t"


class Reducer(object):

    def __init__(self, stream, sep=SEP):
        self.stream = stream
        self.sep    = sep

    def emit(self, key, value):
        sys.stdout.write("{}{}{}\n".format(key, self.sep, value))

    def reduce(self):
        for current, group in groupby(self, itemgetter(0)):
            total = 0
            count = 0

            for item in group:
                total += item[1]
                count += 1
                self.emit(current, float(total) / float(count))

    def __iter__(self):
        for line in self.stream:
            try:
                parts = line.split(self.sep)
                yield parts[0], float(parts[1])
            except:
                continue

if __name__ == '__main__':
    reducer = Reducer(sys.stdin)
    reducer.reduce()
```

`Reducer` 类与 `mapper` 类非常相似，但我们在这段代码中引入了一些新的项目——一个内存安全的迭代器帮助函数 `groupby` 和一个操作符函数 `itemgetter`。和 `mapper` 一样，在 `reduce` 函数中也使用一个迭代器遍历整个数据集，分割键和值。在本例中，键和值是使用 `separator` 分割的。键是由分隔符拆分的第一个项目，使用 Python 切片；值是第一个分隔符之后的其他所有内容。这与 Hadoop Streaming 处理 map 任务的输出时的默认行为相匹配，即将第一个制表符前面的所有字符作为键，剩下的都作为值。由于本例使用了浮点除法来计算平均值，所以简单地将字符串值解析为浮点数。

 在 Python 代码中考虑错误处理是极其重要的。比如，如果输入数据损坏了（不是浮点数或不可分

析），那么将值解析成浮点数时极易发生 ValueError 异常。请注意，异常处理在处理大数据集时至关重要。一种常用的策略是跳过引发异常的行，因为还有大量的数据需要计算。

经过 Hadoop 流水线中的 shuffle 阶段和 sort 阶段之后，进入 reducer 的数据的键是按字母顺序排序的，所以我们希望自动将键及其值组合在一起。groupby 方法以内存安全的方式实现了这一目标，让你能访问键，并像访问列表一样访问值。内存之所以是安全的，是因为 groupby 返回的不是一个保存在内存中的列表，而是一个迭代器，并且一次只读取一行（因此确保大数据集不会让 worker 节点的资源容量不堪重负）。itemgetter 函数简单地指定了应该根据每个元组的哪个值进行分组——在本例中，是元组的第一个元素。

在将值进行高效的内存分组之后，简单地将延误时间相加，除以航班数，然后发射机场作为键、平均值作为值的输出。虽然代码变得冗长了，但希望创建微框架的过程能更加清晰，微框架能消除这段代码中的大部分重复代码，并使其可重用。

3.1.2 执行Streaming作业

在介绍如何通过将作业提交到作业客户端从而在 Hadoop 集群上执行 Streaming 作业之前，先来看一个不产生 Hadoop 集群开销的脚本测试高招。因为 Streaming 使用 Unix 标准管道，所以可以使用 Linux 管道和 sort 命令模拟 Hadoop MapReduce 流水线。

要测试代码，请先确保 mapper.py 和 reducer.py 是可执行的。只需在终端中使用 chmod 命令，如下所示：

```
hostname $ chmod +x mapper.py
hostname $ chmod +x reducer.py
```

要通过 CSV 文件作为输入来测试 mapper 和 reducer 的话，可以使用 cat 命令输出文件的内容，通过管道将输出从 stdout 传输到 mapper.py 的 stdin，再传输到 sort，然后到 reducer.py，最后将结果打印到屏幕上。要测试上一节中计算每个机场平均延误时间的 mapper 和 reducer 的话，可以在终端中执行以下命令，其中 mapper.py、reducer.py 和 flights.csv 都在当前工作目录中：

```
hostname $ cat flights.csv | ./mapper.py | sort | ./reducer.py
ABE     -3.57142857143
ABI     55.375
ABQ     3.83333333333
ABR     -4.0
ABY     -1.33333333333
ACT     -8.2
ACV     109.142857143
ACY     -8.0
ADQ     -14.0
AEX     -6.55555555556
AGS     31.4
ALB     -1.5
ALO     -8.5
AMA     0.8
...
TWF     -7.0
TXK     -4.66666666667
TYR     -6.71428571429
TYS     12.9583333333
VEL     -7.5
VLD     -5.0
VPS     5.06666666667
WRG     -3.75
XNA     14.2580645161
YAK     -17.5
YUM     -0.222222222222
```

Unix 管道是一种测试 Hadoop Streaming 的 mapper 和 reducer 的方法，既简单又有效，还能有效说明集群是如何使用 mapper 和 reducer 代码的。这种方法非常适合在编写脚本时进行快速测试，因为你不用等待 Hadoop Streaming 作业完成，也不需要解析 Java 调用过程（traceback）。如果你在进行测试驱动开发（这是敏捷数据科学的自然补充），则可以使用 `Popen` 模拟管道进行集成测试。

>  在下面的示例中，我们使用 `$HADOOP_HOME` 之类的环境变量指定特定的路径或配置。尽管这些环境变量的名称在每个 Hadoop 发行版中可能有所不同，但它们通常在发行版安装时就已经被设置好了。本书示例假设你使用的是伪分布式的节点设置，如附录 A 所述。

为了将代码部署到集群，需要将 Hadoop Streaming JAR 提交给作业客户端，并传入自定义的操作符参数。Hadoop Streaming 作业的位置取决于 Hadoop 集群的设置。现在假设你设置了环境变量 `$HADOOP_HOME` 并且 `$HADOOP_HOME/bin` 在 `$PATH` 中，`$HADOOP_HOME` 指定了 Hadoop 的安装位置。这样就可以按照

如下所示的方法在集群上执行 Streaming 作业：

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input flights.csv \
    -output average_delay \
    -mapper mapper.py \
    -reducer reducer.py \
    -file mapper.py \
    -file reducer.py
```

请注意，这里使用了 `-file` 选项，它让 Streaming 作业往集群上发送脚本（否则程序无法在节点上找到这些脚本）。执行此命令将在 Hadoop 集群上启动该作业。mapper.py 脚本和 reducer.py 脚本将在处理之前被发送到集群中的每个节点，并应用于流水线的每个阶段。

如果有需要与作业一起发送的其他文件（如航空公司 ID 的查找表），可以使用 `-file` 选项将它们与作业打包在一起。代码中使用的任何第三方依赖也应与作业一起被提交，通常打包在 Python ZIP 文件中。对于较大的依赖文件（例如 NLTK）或者需要使用 Cython 编译的依赖（例如 NumPy 或 SciPy），则需要在作业启动之前在每个节点的系统路径中安装相应依赖。

Hadoop Streaming 有许多其他设置，允许用户指定 Hadoop 库中的类作为 partitioner、输入和输出格式等。但是 Hadoop Streaming 也可以使用 Python 脚本作为 combiner，这在大数据分析中尤为重要。只需使用 `-combiner` 选项即可指定 combiner。一种方法是将 mapper 更新为流水线，使用相同的 shell 脚本、`sort` 和 reducer，和在本地测试脚本一样。不过，指定另一个 Python 脚本作为 combiner 通常更有效，因为大多数 combiner 都与 reducer 完全相同或非常相似。

3.2 Python的MapReduce框架

Hadoop Streaming 稍微高级一点的用法是利用标准错误流（`stderr`）更新 Hadoop 状态以及 Hadoop 计数器。这种技术本质上是让 Streaming 作业访问 `Reporter` 对象——MapReduce Java API 的一部分，用于跟踪作业的全局状态。通过将特殊格式的字符串写入 `stderr`，mapper 和 reducer 可以更新全局作业状态，以报告进度并表明它们是活动的。对于需要大量时间的作业（尤其是涉及从作业的 pickle 文件中加载大型模型的任务），确保框架不会认为任务已超时至关重要。

计数器在整个 MapReduce 框架或应用程序范围内进行全局聚合，以键值对的形式保存数值。这在许多任务中都非常有用，能使分析人员和开发人员了解系统在数据分析期间发生了什么。计数器可以通过满足结合律的运算来累加，这本质上就增加了计数器的值。虽然 Hadoop 实现了多个计数器，能对处理的记录和字节数进行计数，但是自定义计数器能更轻松地跟踪作业中的指标数据或提供副计算的相关渠道。

例如，我们可以在简单的 WordCount 程序中实现计数器，以记录全局单词数以及我们的词汇量（即不同单词的个数），从而进行最终计算——词汇多样性。词汇多样性是单词个数与词汇量的比率，表示单个单词在文本中的平均出现频率。这类指标数据对于理解语料库的变化对自然语言处理应用程序的影响至关重要。副计算指标数据，也就是这里的计数器，追踪 Hadoop 作业的主体，可以用作输出，但不影响主要的计算。当评估跨数据集的机器学习模型时，也可以用它们来计算均方误差或分类指标。

要使用 `Reporter` 的 `Counter` 和 `Status` 功能的话，可以为上一节中的 `Mapper` 和 `Reducer` 类添加如下方法：

```
def status(self, message):
    sys.stderr.write("reporter:status:{}\n".format(message))

def counter(self, counter, amount=1, group="ApplicationCounter"):
    sys.stderr.write(
    "reporter:counter:{},{},{}\n".format(group, counter, amount)
    )
```

`counter` 方法允许 `map` 和 `reduce` 函数更新任意命名计数器的计数。根据需要，更新的值可为任意值（默认为递增 1）。可以将计数器的组设置为任意名称，通常默认为应用程序的名称。与之类似，`status` 方法允许 MapReduce 应用程序向框架发送任意消息，并使它们在日志或 Web 用户界面中可见。

为了扩展航班平均延误应用程序，让它提供准点和延迟航班的计数，并在开始和结束时发送状态更新，按如下所示更新 `map` 函数：

```
def map(self):
    self.status("mapping started")
    def map(self):
        for row in self:
            if row[6] < 0:
                self.counter("early departure")
```

```
            else:
                self.counter("late departure")

            self.emit(row[3], row[6])

    self.status("mapping complete")
```

仅仅通过添加这几行，我们就能更好地了解平均延误程序是如何运行的，而不需要专门编写冗长的 Hadoop 作业计算准点和晚点航班的计数。我们可能会想在 reducer 中计算有多少机场有航班数据。因为 reducer 将看到数据集中的每个机场，因此可以这样更新 `reduce` 函数：

```
def reduce(self):
    for current, group in groupby(self, itemgetter(0)):
        self.status("reducing airport {}".format(current))
        ...

        self.counter("airports")
        self.emit(current, float(total) / float(count))
```

随着分析应用程序规模的增长，这些能实现 Hadoop Streaming 全部功能的技术将变得至关重要。再以自然语言处理为例，为了进行词性标注或命名实体识别，应用程序必须将 pickle 之后的模型加载到内存中。此过程可能持续几秒钟到几分钟——使用状态机制警告框架任务仍在正常运行，可以确保集群不会因为推测执行机制而瘫痪。即使在运行其他作业时，计数器也能帮助应用程序从全局范围分析大型数据集。

既然提到了全局范围，就来说说最后一个能改进由 Python 编写的 Streaming 应用程序的工具：作业配置变量（job configuration variable，简称“JobConf 变量”）。Hadoop Streaming 应用程序自动将作业的配置变量添加到环境中，用下划线（_）替换圆点（.）来重命名配置变量。例如，如果要访问作业中 mapper 的数量，可以请求 `"mapred.map.tasks"` 配置变量。虽然这个示例不一定有用，但用户定义的配置值可以使用圆点形式的 `-D` 参数提交到 Hadoop Streaming 中，其中可包含重要信息，如共享资源的 URL。要在 Python 代码中访问作业配置变量，可添加以下函数：

```
import os

def get_job_conf(name):
```

```
    name = name.replace(".", "_").upper()
    return os.environ.get(name)
```

很明显，使用一个微型、可重用的框架对 Hadoop Streaming 的 Python 开发大有帮助。该框架应该有一个用于处理 mapper 和 reducer 的 Streaming 细节的基类，以及应该在自定义 MapReduce Streaming 作业中扩展的抽象基类 `Mapper` 和 `Reducer`。想想下面的框架：

```
import os
import sys

from itertools import groupby
from operator import itemgetter

SEPARATOR = "\t"


class Streaming(object):

    @staticmethod
    def get_job_conf(name):
        name = name.replace(".", "_").upper()
        return os.environ.get(name)

    def __init__(self, infile=sys.stdin, separator=SEPARATOR):
        self.infile = infile
        self.sep = separator

    def status(self, message):
        sys.stderr.write("reporter:status:{}\n".format(message))

    def counter(self, counter, amount=1, group="Python Streaming"):
        msg = "reporter:counter:{},{},{}\n".format(group, counter, amount)
        sys.stderr.write(msg)

    def emit(self, key, value):
        sys.stdout.write("{}{}{}\n".format(key, self.sep, value))

    def read(self):
        for line in self.infile:
            yield line.rstrip()

    def __iter__(self):
        for line in self.read():
            yield line


class Mapper(Streaming):
```

```
    def map(self):
        raise NotImplementedError("Mappers must implement a map method")


class Reducer(Streaming):

    def reduce(self):
        raise NotImplementedError("Reducers must implement a reduce metho

    def __iter__(self):
        generator = (line.split(self.sep, 1) for line in self.read())
        for item in groupby(generator, itemgetter(0)):
            yield item
```

在编写传递给 Hadoop Streaming 的 mapper 和 reducer 时，只需在 Streaming 作业中引入这个文件，并从该框架中导入合适的类。在扩展类后，只需在代码中实现 `map` 函数或 `reduce` 函数即可。下一节描述了一个具体的示例——将此框架与自然语言工具包（natural language toolkit，NLTK）结合使用，以执行更精确的单词计数。

3.2.1　短语计数

一直以来，Hadoop 程序的“Hello, World”都是单词计数程序。使用 Python 代码对文件执行单词计数是演示分布式计算的好方法，但是有了 Hadoop Streaming 的话，就可以简单地使用 Linux 的 `wc` 命令，因为这条命令也从 `stdin` 接收输入，并输出到 `stdout`。通过使用 mapper 和 reducer，Python 允许我们使用多个 reducer 任务进行数据聚合，将处理极大数据集的工作分解开来。此外，单词计数是语言处理所用的统计方法的基础，我们可以使用 Python 中可用的高级文本处理技术来进行更高级的词法分析，例如使用词形还原（lemmatization）技术进行短语计数或更高级的创建索引的方法。

Hadoop Streaming 非常适用于文本处理，不仅因为通过它能访问 TextBlob 和 NLTK 等库，还因为 Hadoop Streaming 本身就以逐行方式使用字符串序列。默认情况下，Hadoop Streaming 期望通过 Streaming 作业的标准输入和标准输出的文本是由制表符分隔的——如果你还能将数据视为键值对，那当然更好，不过也不是必须的。

> NLTK 和 TextBlob 是第三方依赖，这意味着 Python 默认不包含它们。要安装这些包，可以使用 Python 包管理器 `pip`，并且集群中的每个节点都必须安装这些依赖。为了简化问题，假定所有额外的库都已经安装在集群上，不过集群管理不在本书的范围

之内。如果你使用的是伪分布式设置，那么执行 `pip install nltk` 应该就能完成 NLTK 的安装。

使用 Python 微框架来编写一个稍加改进的 MapReduce 应用程序，执行单词计数。首先，将单词全部归范化为小写字母，像“Apple”这样的单词将与“apple”相同。不是所有语言处理应用程序都可以这么做——英语（和许多其他语言）的大小写是语法的重要部分，表明句子开始或专有名称（例如 Apple Paltrow 或 Apple, Inc.）。然而，在词汇评估中采用规范化倒是没什么问题，Apple 在句子开头还是作为专有名词在这里无关紧要。

下一步，从单词计数中去掉标点符号和停用词。停用词是语言中的虚词，例如冠词（“a”或“an”）、限定词（“the”“this”“my”）、代词（“his”“they”）和介词（“over”“on”“for”）。由于停用词具备这种功能性用法，所以它们的出现非常频繁，占据了语料库的一大部分。据说，对一些信息检索应用程序来说，停用词是给定词汇分布中最常见的词，所以被自动去除以提高应用程序的性能。在这种情况下，“want”或“has”这样的普通动词可能会被排除在外。不管是去除停用词、标点符号，还是文本的标准化，都可以大幅降低词汇密度，对了解特定语料库中最重要的单词大有帮助。

最后，使用这个归范化的语料库来统计短语的个数，也就是经常一起出现的单词（例如忽略停用词后连续两次一起出现）。统计学家通过短语分析来获取通常一起出现或可能具有某种特殊意义的词语，例如“lawn chair”（草坪躺椅）或“vetoed bill”（否决议案）。短语是 *n*-Gram 语言模型中最简单的形式。*n*-Gram 是一种模型构建技术，可以预测给定上下文的下一个单词。

使用之前的框架，`Mapper` 完成了大部分工作：

```
#!/usr/bin/env python

import sys
import nltk
import string

from framework import Mapper

class BigramMapper(Mapper):

    def __init__(self, infile=sys.stdin, separator='\t'):
        super(BigramMapper, self).__init__(infile, separator)

        self.stopwords = nltk.corpus.stopwords.words("english")
```

```
        self.punctuation = string.punctuation

    def exclude(self, token):
        return token in self.punctuation or token in self.stopwords

    def normalize(self, token):
        return token.lower()

    def tokenize(self, value):
        for token in nltk.wordpunct_tokenize(value):
            token = self.normalize(token)
            if not self.exclude(token):
                yield token

    def map(self):
        for value in self:
            for bigram in nltk.bigrams(self.tokenize(value)):
                self.counter("words") # 计算短语的总数
                self.emit(bigram, 1)


if __name__ == "__main__":
    mapper = BigramMapper()
    mapper.map()
```

map 方法很直观。它循环遍历整个输入，并使用 nltk.wordpunct_tokenizer 对值进行分词（tokenize），值是数据集的一行文本。这个分词器确保不仅单词（包括缩写）会被拆分，标点符号也会被拆分。通过使用 nltk 内置的停用词语料库，分词器还能忽略标点符号和停用词。请注意，使用自定义停用词列表改进此代码并不难，可以使用 -file 参数将该列表与作业一起打包。

为了执行短语计数，需要发射键为令牌、值为 1 的元组。我们创建了一个简单的 emit 帮助函数，它将由分隔符字符串分隔的键值对写入到输出（在本例中是 stdout）。可以使用内置的 nltk.bigrams 函数收集短语。

reducer 实现了一个非常常见的 MapReduce 模式——SumReducer。这类 reducer 的使用频率非常高，你甚至都想把它当作一个标准类，与 IdentityMapper 和其他标准模式（你会在第 5 章见到这些模式）一道添加到微框架中。SumReducer 的代码如下所示：

```
#!/usr/bin/env python

from framework import Reducer
```

```
class SumReducer(Reducer):

    def reduce(self):
        for key, values in self:
            total = sum(int(count) for count in values)
            self.emit(key, total)


if __name__ == '__main__':
    reducer = SumReducer()
    reducer.reduce()
```

请注意，这个 reducer 忽略了键——文本字符串形式的短语元组，因为 reducer 只计算该元组的出现次数。然而，如果为了改变键空间（例如基于第一个单词过滤短语）而需要处理这个复合键，或者因为复合键在链式 MapReduce 作业的一个 mapper 中而需要处理这个复合键，可以使用 Python 的 `literal_eval` 将字符串转换为元组：

```
import ast

key = ast.literal_eval(key)
```

使用与之前示例相同的命令，通过 Hadoop Streaming 将此作业提交到集群，但要确保 framework.py 文件也被打包并随作业一起发送。作业提交命令如下所示：

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input corpus \
    -output bigrams \
    -mapper mapper.py \
    -reducer reducer.py \
    -file mapper.py \
    -file reducer.py \
    -file framework.py
```

请注意，在这个例子中，假设第三方依赖 `nltk` 已被经安装在集群中的每个节点上——如果你有集群的管理访问权限，或者能从特定的 AMI 启动集群，就可以做到。否则，`nltk` 需要打包成 ZIP 文件，并使用 `-file` 参数发送到集群。

3.2.2　其他框架

虽然我们已经创建了一个可以编写 MapReduce 作业的小框架，但还有其他一些框架也能让你使用 Python 编写 MapReduce 作业，这点十分重要。编写本书时，最流行的两个框架是 Yelp 的 mrjob（http://bit.ly/1NqmsvA）和 GitHub 上的 dumbo（http://bit.ly/1UQx8G3），它们封装了 Hadoop Streaming 并添加了更多的功能。其他框架还有封装了 Hadoop Pipes（Hadoop 的 C++ API）的 pydoop（http://bit.ly/1LizcVD）和使用 Cython 封装了 Streaming 的 hadoopy（http://bit.ly/1TQugXl）。

这些框架试图帮助 Python 开发人员编写 Hadoop 作业，但是却造成了性能损失。它们提供了更简单的 API、编程接口以及 Python 中的标准工具，甚至还提供了运行和启动作业的方法，让开发人员能更专注于 Python 开发而不是 Hadoop 集成。更高级的框架有 `TypedBytes`，这是一种 Hadoop 中的二进制序列化格式，支持将 Python 对象序列化为输入和输出，能显著提高这些框架的性能。

`mrjob` 库也值得关注一下，因为 Yelp 正在积极开发它，其平台完全在 Amazon Web Services 生态系统内。因此，`mrjob` 是唯一适合通过 Python 的 `boto` 库在 Amazon 的 Elastic MapReduce 框架上快速部署和运行作业的库。使用 `mrjob` 编写的作业通常是包含完整 MapReduce 代码的单个文件，可以直接在本地文件系统、EMR 或常规的 Hadoop 集群上执行。此外，可以通过简单的配置文件配置作业。

`dumbo` 库是最早的 Python Hadoop Streaming 框架之一，虽然没有被经常维护，但使用广泛。Tom White 编写的《Hadoop 权威指南》认为它是可选的框架。`dumbo` 框架完全封装了 Hadoop Streaming，并使用 `TypedBytes` 来提高性能。通过它，可以高效地编写复杂的链式 MapReduce 作业。它还附带用于管理和执行作业、提供与 HDFS 交互的命令行脚本。

最后，简单地使用 Hadoop Streaming 是迄今为止性能最优的解决方案，因为它不依赖于第三方库，而且足够轻巧，可以部署在各种分析场景中。本书中的 MapReduce 示例将使用本章描述的 Streaming 机制。对于更大、更复杂的分析，开发人员需要评估这些框架，使其成为工作流程的一部分。

3.3 MapReduce进阶

这一节将介绍一些与 MapReduce 紧密相关的高级主题，引入一些在

MapReduce 算法和优化中发挥重要作用的概念，因为你在阅读其他有关如何实现不同分析的资料时会遇到这些术语。这里不会介绍如何使用这些工具，而是从概念层面去介绍，以便在你对 MapReduce 进行深入探索时，不会对它们感到陌生。

这些工具很难在没有 Java API 的情况下实现，因此不适合将它们放入介绍 Hadoop Streaming 的章节中，但是在讨论 MapReduce 时对它们避而不谈又实在是说过不去。我们将在后文讨论 combiner（主要的 MapReduce 优化技术）、partitioner（确保在 reduce 步骤中不出现瓶颈的技术）和作业链（用于组合更大的算法和数据流的技术）。

3.3.1 combiner

mapper 会产生大量的中间数据，这些中间数据必须通过网络传输，进行 shuffle、sort 和 reduce。由于网络是物理资源，大量数据的传输可能会导致作业延迟和内存瓶颈（比如 reducer 要保存到内存中的数据太多）。combiner 是解决这个问题的主要机制，而且它本质上也是与 mapper 输出相关联的中间 reducer。在将数据转发到合适的 reducer 之前，combiner 通过执行一个 mapper 局部的 reduce 来减少网络流量。例如，两个 mapper 和一个简单的求和 reducer 产生如下输出。

mapper 1 的输出：

```
(IAD, 14.4), (SFO, 3.9), (JFK, 3.9), (IAD, 12.2), (JFK, 5.8)
```

mapper 2 的输出：

```
(SFO, 4.7), (IAD, 2.3), (SFO, 4.4), (IAD, 1.2)
```

求和 reducer 的目标输出：

```
(IAD, 29.1), (JFK, 9.7), (SFO, 13.0)
```

每个 mapper 都为 reducer 带来额外的工作，即每个 mapper 都会产生重复的键。combiner 预先计算每个键的和，减少生成的键值对的数量，从而减少网络流量。此外，因为存在较少的重复键，所以

shuffle 操作和 sort 操作也变得更快。

只要运算满足交换律和结合律，combiner 和 reducer 就是相同的——这很常见，但也不总是这样。只要 combiner 的输入输出数据类型和 mapper 的输出数据类型一样，则 combiner 可以执行任意的局部聚合（partial reduction）。因此，combiner 的运算与 reducer 不同时，算法常常同时使用 mapper、reducer 和 combiner 实现。要在 Hadoop Streaming 中指定 combiner，可以使用 `-combiner` 选项，与指定 mapper 和 reducer 类似：

```
$ hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-*.jar \
    -input input_data \
    -output output_data \
    -mapper mapper.py \
    -combiner combiner.py \
    -reducer reducer.py \
    -file mapper.py \
    -file reducer.py \
    -file combiner.py
```

如果 combiner 与 reducer 匹配，则只需将 reducer.py 文件指定为 combiner 即可，无须添加额外的 combiner 文件。在我们创建的微框架中，`combiner` 类会简单地继承 `Reducer`。

3.3.2 partitioner

partitioner 通过划分键空间来控制如何将键及其值发送到每个 reducer，默认使用的 `HashPartitioner` 通常就能满足需求。它通过计算键的散列值并将键分配给由 reducer 数量确定的键空间，来将键均匀地分配给每个 reducer。给定均匀分布的键空间后，每个 reducer 将获得相对平均的工作负载。

一旦键的分布不平均，比如大量的值与一个键相关联，其他键几乎没有关联的值，问题就出现了。在这种情况下，大部分 reducer 的工作量不饱满，并行 reduce 的大多数好处也就无从体现。一个自定义的 partitioner 可以根据散列之外的其他语义结构（通常是特定于领域的）划分键空间，从而缓解这个问题。某些类型的 MapReduce 算法也可能需要自定义 partitioner，最典型的就是实现左外连接。最后，因为每个 reducer 都将输出写入自己的 part-* 文件，使用自定义 partitioner 还能支持更清晰的数据组织，让你根据分区条件将分段输出写入每个文件，例如写入按年输出数据。

不幸的是，只能使用 Java API 创建自定义 partitioner。不过 Hadoop Streaming 用户仍然可以从 Hadoop 库中指定 partitioner Java 类，或者编写自己的 Java partitioner 并将其与 Streaming 作业一起提交。

3.3.3 作业链

大多数复杂的算法不能使用简单的 `map` 和 `reduce` 描述。因此，为了实现更复杂的分析，需要一种被称为作业链的技术。如果可以将复杂的算法分解成几个较小的 MapReduce 任务，那么将这些任务链接在一起就可以产生完整的输出。考虑计算数据集中变量对的 Pearson 相关系数。要得到 Pearson 相关系数，需要计算每个变量的平均值和标准差。采用单个 MapReduce 无法轻松实现这个目标，所以可以采用以下策略。

(1) 计算每一个 (X, Y) 的平均值和标准差。

(2) 使用第一个作业的输出来计算协方差和 Pearson 相关系数。

平均值和标准差可以在初始作业中计算：在 mapper 中计算总个数、和以及平方和，然后在 reducer 中计算平均值和标准差。第二个作业取第一个作业输出的平均值和标准差，在 mapper 中将各个值和平均值的差值相乘来计算协方差，然后通过适当求和和取平方根来计算出该相关系数。如图 3-2 所示，第二个作业依赖于第一个作业。

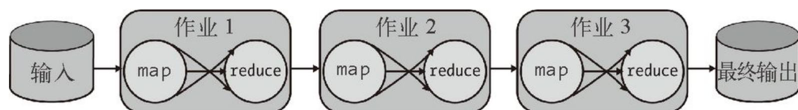


图 **3-2**：线性作业链将一个或多个 **MapReduce** 作业的输出作为输入发送到另一个作业来产生完整的计算

因此，作业链是许多小作业的组合，通过将一个或多个作业的输出发送给另一个作业作为输入，从而实现完整的计算。为了实现这样的算法，开发人员必须考虑每一步计算怎样 reduce 出中间值——不仅仅是 mapper 和 reducer 之间的中间值，还包括作业之间的中间值。如图 3-2 所示，许多作业都被认为是线性作业链。线性依赖性意味着每个 MapReduce 作业仅依赖于前一个作业。然而，这是一种简化的作业链形式，更普遍的作业链表示为作业依赖于一个或多个先前作业的数据流。可以用有向无环图（directed acyclic graph，DAG）来表述复杂作业，它描述数据如何从输入源通过每个作业（有向部分）流向

下一个作业（从不重复步骤，无环部分），最后作为最终输出（如图 3-3 所示）。

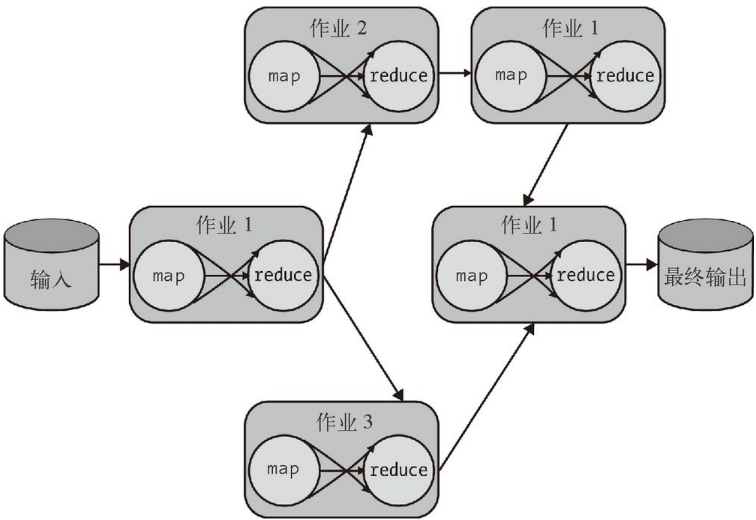


图 **3-3**：数据流作业链是线性作业链的扩展

仅有 **map** 的作业

在考虑作业链和作业依赖时，注意可能会有仅有 map 的作业。仅有 map 的作业分两种：不需要聚合的作业，以及积极避免 shuffle 阶段和 sort 阶段的作业——要么为了保持数据的顺序，要么为了优化作业的执行。

要执行仅有 map 的作业，只需将 reducer 的数目设置为 0 即可。有了 Hadoop Streaming，就可以使用 `-numReduceTasks` 标志指定 reducer 的数目。通过使用 identity mapper，也可以实现仅有 reduce 的作业，第 5 章会讨论这个问题。“仅有 map”的作业可以使用 identity reducer 实现排序。

为了计算 Pearson 相关系数并说明作业链，我们将使用由维基百科给出的公式计算样本中的 Pearson 相关系数。输入的数据是键值对，其中键是因变量 y，值是因变量的向量，要求变量 x_i 与 y 的相关性。

等式 3-1 计算样本的 Pearson 相关系数的公式

$$r = r_{xy} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

第一个 MapReduce 作业将计算 n 以及 x 和 y 的平均值，如下所示：

```
class VariablePairsMapper(Mapper):

    def map(self):
        # 计算(x, y)的个数及和
        # 输出键是x在vector中的索引
        for y, vector in self:
            for x, i in enumerate(vector):
                self.emit(i, (1, x, y) )


class PairsMeanReducer(Reducer):

    def reduce(self):
        for key, values in self:
            # 将所有的值加载到内存，这样可以迭代两次
            values = list(values)

            # 计算(x, y)的和及元素的个数
            sx, sy, sn = 0
            for (n, x, y) in values:
                sn += n
                sx += x
                sy += y

            # 计算x和y的平均值
            xbar = sx / n
            ybar = sy / n

            # 发射每一个(x, y)及x和y的平均值
            for (n, x, y) in values:
                self.emit(key, (x, y, xbar, ybar))
```

有了平均值之后，就可以计算协方差和标准差，以便在第二个作业中计算 Pearson 相关系数。为此，需要再次传入相同的输入数据，并将第一个作业的输出作为该作业的输入。为了简化以说明问题，将每个 (x, y) 及其相关联的均值作为第一个作业的输出：

```
import math


class PearsonMapper(Mapper):

    def map(self):
```

```
        # 计算x和xbar的差及y和ybar的差
        # 发射差的乘积及其平方
        for i, (x, y, xbar, ybar) in self:
            xdiff = x-xbar
            ydiff = y-ybar
            self.emit(i, (xdiff*ydiff, xdiff**2, ydiff**2))


class PearsonReducer(Reducer):

    def reduce(self):
        for key, values in self:
            # 计算差乘积的和以及平方的和
            sxyd = 0
            sxd2 = 0
            syd2 = 0

            for (xyd, x2d, y2d) in values:
                sxyd += xyd
                sxd2 += x2d
                syd2 += y2d

            # 发射相关系数
            r = sxyd / (math.sqrt(sxd2) * math.sqrt(xyd2))
            self.emit(key, r)
```

虽然这不是实现并行计算 Pearson 相关系数的最有效方法，但它展示了一个清晰的作业链。首先要考虑的问题是，如何输出作业 1 的数据，才能让作业 2 运行。第 5 章将从这个简单的引子深入到更复杂的主题，探索 pair 和 stripe，让这样的复杂计算变得更可行。虽然可以人工设置作业链（在命令行上先执行第一个作业，然后执行第二个作业），但这不是最理想的办法。在第 8 章中，我们将探讨数据流，以及如何使用高级工具将作业链接起来。

3.4　小结

Hadoop Streaming 是重要的工具，让使用 R 或 Python（而不是 Java）编程的数据科学家能够立即开始使用 Hadoop，特别是 MapReduce。长久以来，如果你想使用 Python 处理大数据，除了 Hadoop Streaming 别无他法。但对于更复杂的作业或算法，特别是那些需要使用 combiner 或 partitioner 进行优化的作业，就需要使用 Java API 了。

然而此时，事情出现了转机，特别是对 Python 开发人员来说。下一章将讨论 Spark——一个与众不同的 Hadoop 计算框架，附带了原生

的 Python API（很快就会有 R API）。Spark 正迅速成为数据科学平台的首选，很大一部分是因为 DataFrame 和大量分析包等工具正在 Spark 上构建。

但是，MapReduce 和 Hadoop Streaming 没有完全被 Spark 包含。实际上，如果仅仅就内置的 shuffle 和 sort 而言，其实更适合用 MapReduce 处理批处理作业，特别是那些经常运行的作业（例如 ETL 操作或其他数据整理和清理过程）。此外，MapReduce 和 Hadoop Streaming 经过精心构建与层层测试，可以为任务关键型应用程序所信任。事实上，因为编程模型间太过相似，现在大多数的大数据会同时使用 MapReduce 和 Spark，二者都能很好地满足其特定类型的应用程序。

第 4 章　Spark 内存计算

在过去十年中，HDFS 和 MapReduce 一直是大规模机器学习、大规模分析和大数据设备的基石和驱动力。与大多数平台技术一样，Hadoop 的成熟已经带来了一个稳定且通用的计算环境，足以为图形处理、微批处理、SQL 查询、数据仓储和机器学习等任务构建专用工具。然而，Hadoop 越普及，越要求它具备能应对更广泛的新场景的专业化能力。并且我们越发觉得 MapReduce 的批处理模型不太适合常见的工作流，包括针对单个数据集的迭代、交互和按需计算。

主要的 MapReduce 抽象（将计算规定为 map 和 reduce）是并行的，它易于理解，并且隐藏了分布式计算的细节，从而保证 Hadoop 的正确性。然而，为了实现协调性和容错性，MapReduce 模型使用拉执行模型，需要将中间数据写回 HDFS。不幸的是，将数据从其存储位置移动到计算位置所需要的 I/O 在任何计算系统中都是最大的时间成本。因此，MapReduce 在具有极高的安全性和弹性的同时，运行任务的速度也不可避免会慢一些。更糟的是，几乎所有应用程序都必须在多个步骤中将多个 MapReduce 作业链接在一起，从而创建面向最终所需结果的数据流。这导致不为用户所需的大量中间数据被写入 HDFS，从而产生额外的磁盘开销。

为了解决这些问题，Hadoop 采用了更通用的资源管理框架进行计算，这便是 YARN。以前，MapReduce 应用程序分配给作业的资源（处理器和内存）只能被 mapper 和 reducer 使用，而 YARN 为

Hadoop 应用程序提供了更通用的资源访问。因此，专用工具不再需要分解为一系列 MapReduce 作业，可以变得更复杂。通过泛化集群管理，可以扩展最初设想的 MapReduce 编程模型，以包括新的抽象和操作。

Spark 是应运而生的第一个快速、通用的分布式计算范式，并且因其速度和适应性而迅速得到了普及。Spark 主要通过名为弹性分布式数据集（resilient distributed dataset，RDD）的新数据模型实现高速运行。该数据模型在计算时存储在内存中，从而避免了昂贵的中间磁盘写操作。它还利用了 DAG 执行引擎优化计算，特别是迭代计算，这对于优化算法和机器学习等数据理论任务来说至关重要。在速度方面的优势使得 Spark 能以交互方式进行访问（就像访问 Python 解释器一样），使用户成为计算任务的一部分，并支持以前不可能实现的大数据集探索，让数据科学家能更轻松地使用集群。

> 因为有向无环图通常用于描述数据流中的步骤，所以在讨论大数据处理时经常会使用 DAG 这一术语。DAG 有向，是因为一个或多个步骤接着前一个步骤；无环，是因为单个步骤不重复。当数据流被描述为 DAG 时，它消除了大代价的同步，并且使得并行应用程序更容易构建。

本章将介绍 Spark 和 RDD，也是讲解使用 Hadoop 进行分析的基础知识的最后一章。因为 Spark 实现了数据科学家熟悉的许多应用程序（例如 DataFrame、交互式 notebook 和 SQL），所以建议 Hadoop 新手用户首选 Spark 与集群交互，至少初期要这样做。为此，我们将描述 RDD，通过 `pyspark` 探索如何在命令行中使用 Spark，然后演示如何使用 Python 编写 Spark 应用程序，并将它们作为 Spark 作业提交到集群中。

4.1　Spark基础

Apache Spark 是一个集群计算平台，为类似于 MapReduce 模型的分布式编程提供了一个 API，但被设计用于快速的交互式查询和迭代算法。1 它主要通过在集群节点的内存中缓存计算所需的数据来实现高速运行。在内存中进行集群计算使 Spark 可以运行迭代算法，因为程序可以为数据创建检查点并引用回它，避免从磁盘重新加载。此外，它支持极快速的交互式查询和流式数据分析。因为 Spark 与 YARN 兼容，所以它可以在现有的 Hadoop 集群上运行并访问任何 Hadoop 数据源，包括 HDFS、S3、HBase 和 Cassandra。

> 1《Spark 快速大数据分析》，Holden Karau、Andy Konwinski、Patrick Wendell、Matei Zaharia 著。本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1558。

还有一点很重要，Spark 的设计从根本上支持大数据应用程序和数据科学任务。Spark API 不仅支持 `map` 和 `reduce`，还提供了许多强大的分布式抽象。这些抽象同样与函数式编程相关，包括 `sample`、`filter`、`join` 和 `collect` 等。此外，虽然 Spark 是用 Scala 实现的，但是 Scala、Java、R 和 Python 的编程 API 使得许多数据科学家能更轻松地使用 Spark，并能够快速且充分地利用 Spark 引擎。

为了理解这种转变，先来看看 MapReduce 在迭代算法方面的局限性。迭代算法对数据块多次应用相同的运算，直到得到预期的结果。例如，像梯度下降这样的优化算法是迭代的——给定某个目标函数（如线性模型），目标是优化该函数的参数，最小化误差（模型的预测值和数据的实际值之间的差）。算法将拥有一组参数的目标函数应用于整个数据集并计算误差，再根据计算得到的误差（沿误差曲线下降）略微修改函数的参数。重复该过程（即迭代），直到误差小于某个阈值或者直到达到最大迭代次数。

这种基本技术是许多机器学习算法（特别是有监督学习）的基础。在有监督学习中，正确答案是提前已知的，可以用它来优化决策空间。为了使用 MapReduce 编程实现这种类型的算法，目标函数的参数必须映射到数据集中的每个实例，并计算和规约误差。在 reduce 阶段结束之后，参数将被更新并馈送到下一个 MapReduce 作业。可以将误差计算和作业更新链接在一起来实现这一步。然而，每个作业都必须从磁盘读取数据，并将误差写回磁盘，这将导致明显的 I/O 延迟。

相反，Spark 在应用程序运行期间将尽可能多的数据集保存在内存中，从而防止在迭代之间重新加载数据。因此，Spark 程序员不是简单地指定 map 步骤和 reduce 步骤，而是在执行某个需要协调的动作（如规约或写入磁盘）之前，指定一系列将应用于输入数据的数据流转换。因为数据流可以通过 DAG 来描述，所以 Spark 的执行引擎提前知道了如何在集群上分发计算并管理计算的细节，这与 MapReduce 抽象分布式计算的方式相似。

通过结合无环数据流和内存计算，Spark 能达到非常快的速度，特别是当集群大到足以容纳内存中所有数据时。事实上，通过增加集群的大小，使内存可以容纳整个非常大的数据集，Spark 可以支持交互式使用——使用户成为正在集群上运行的分析进程的关键参与者。随着 Spark 的发展，用户交互的概念成为其分布式计算模型的基础。事实

上，很可能就是因为这个原因，Spark 才支持这么多语言。

Spark 的通用性也意味着可以用它构建更高级的工具，用于实现类 SQL 的计算、图形处理和机器学习算法，乃至交互式 notebook 和数据框——这些都是为数据科学家所熟知的工具，但是是在集群环境中实现的。在介绍 Spark 如何实现通用分布式计算的细节之前，先来了解一下它有哪些工具。

4.1.1 Spark栈

Spark 是一种通用的分布式计算抽象，可以在独立模式下运行。但是 Spark 只专注于计算而不关心数据存储，因此通常在实现了数据仓储和集群管理工具的集群中运行。本书主要关注 Hadoop（尽管也有 Apache Mesos 和 Amazon EC2 上的 Spark 发行版）。当使用 Hadoop 构建 Spark 时，它使用 YARN 通过 `ResourceManager` 来分配和管理集群资源，如处理器和内存。正因如此，Spark 可以访问所有 Hadoop 数据源，例如 HDFS、HBase、Hive，等等。

Spark 通过 Spark Core 模块将其主要的编程抽象暴露给开发人员。此模块包含基本功能和常规功能，包括定义 RDD 的 API。我们将在下一节中更详细地介绍 RDD，它是所有 Spark 计算的基本功能。Spark 构建于这个核心之上，为各种数据科学任务实现与 Hadoop 交互的专用库，如图 4-1 所示。



图 4-1：Spark 是一个计算框架，旨在利用集群管理平台（如 YARN）和分布式数据存储（如 HDFS）

因为组件库未集成到通用计算框架中，所以 Spark Core 模块非常灵

活，允许开发人员以不同的方法轻松解决类似的用例。例如，将 Hive 迁移到 Spark 可以为现有用户提供一条简单的迁移路径；GraphX 基于以顶点为核心的图计算模型 Pregel，但其他利用 gather、apply、scatter（GAS）风格计算的图形库可以很轻松地用 RDD 实现。这种灵活性意味着 Spark 不仅可以用来开发专业工具，同时也可以帮助新用户快速上手已经存在的 Spark 组件。

Spark 主要包含如下组件。

Spark SQL

起初用于与 Spark 进行交互的 API，通过 HiveQL（SQL 的 Apache Hive 变体）实现，你现在也仍然可以通过这个库直接访问 Hive。不过这个库也在不断升级，提供更常规、更结构化的数据处理抽象——DataFrame。DataFrame 本质上是组织成列数据的分布式集合，概念上类似于关系数据库中的表。

Spark Streaming

对无界数据流实现实时处理和操作。虽然有许多用于处理实时数据的库（例如 Apache Storm），但是 Spark Streaming 使程序能够像处理一般的 RDD 一样处理实时数据。

MLlib

一个常用的机器学习算法库，使用 RDD 上的 Spark 操作实现。该库包含可扩展的学习算法（例如分类、回归等），这些算法需要进行大型数据集的迭代运算。以前人们选择使用的大数据机器学习库是 Mahout 库，以后将转而使用 Spark。

GraphX

算法和工具集合，用于操作图形、执行并行图形的操作和计算。GraphX 扩展了 RDD API，囊括了操作图形、创建子图和访问路径中所有顶点的操作。

这些组件与 Spark 编程模型结合，提供了大量与集群资源交互的方法。很可能就是因为这种完整性，Spark 才在分布式分析中如此流行。不需要学习多个工具，基本的 API 在组件中保持不变，而且组件无须额外安装便可轻松访问。这种丰富性和一致性源于 Spark 中的主要编程抽象，即弹性分布式数据集（RDD），到目前为止它已被多次提及，下一节将对它进行更详细的探讨。

4.1.2 RDD

在第 2 章中，我们将 Hadoop 描述为一个分布式计算框架，涉及两个主要问题：如何在集群中分发数据，以及如何分发计算。分布式数据存储问题涉及数据的高可用性（将数据放在它被处理的地方）、可恢复性和持久性。分布式计算意在通过将大型计算或任务分解成更小的独立计算来提高计算的性能（速度），这些计算可以同时（并行）运行，然后聚合得到最终结果。因为每个并行计算在集群中单独的节点或计算机上运行，所以分布式计算框架需要为整个计算提供一致性、正确性和容错保证。Spark 不处理分布式数据存储，而是依靠 Hadoop 提供此功能，因此通过一个被称为弹性分布式数据集的框架专注于提供可靠的分布式计算。

RDD 本质上是一种编程抽象，表示跨机器分区的对象的只读集合。RDD 可以根据转换过程（lineage）重建（因此是容错的），通过并行操作访问，从分布式存储（例如 HDFS 或 S3）读取和写入，以及高速缓存在 worker 节点的内存中以快速重用——这一点最为重要。如前所述，这种内存缓存功能使速度大幅提高，并提供了机器学习所需的迭代计算以及以用户为中心的交互式分析。

RDD 使用函数式编程结构体进行操作，函数式编程结构体包括 `map` 和 `reduce`，并在其基础上进行扩展。程序员通过从输入源加载数据，或转换现有集合来创建新的 RDD。**RDD** 转换过程（lineage）主要由应用于 RDD 转换的历史定义，并且因为 RDD 的对象集合是不可变的（不能直接修改），所以转换可以重新应用于部分或整个集合，以便从故障中恢复。因此，Spark API 本质上是创建、转换和导出 RDD 操作的集合。

> Spark 中的故障恢复与 MapReduce 的区别很大。在 MapReduce 中，数据在每个临时处理步骤之间是作为 sequence 文件（保存带类型的键值对的二进制平面文件）写入磁盘的。因此，进程在 map、shuffle 和 sort、reduce 之间拉取数据。如果一个进程失败，那么另一个进程就开始拉取数据。在 Spark 中，对象集合存储在内存中。通过保留 RDD 部分的早期检查点或缓存版本，RDD 转换过程（lineage）可用于重建部分或全部集合。

因此，基本编程模型描述了如何通过编程操作创建和修改 RDD。有两种类型的操作可以应用于 RDD，分别是转换和动作。将转换操作应用于现有 RDD 可以创建新的 RDD，例如对 RDD 应用过滤操作可以生

成包含过滤出来的值的较小 RDD。然而，动作才会真正将结果返回给 Spark 驱动程序，协调或聚合 RDD 中的所有分区。在这个模型中，`map` 是一种转换操作，因为一个函数被传递给存储在 RDD 中的每个对象，并且该函数的输出映射到一个新的 RDD；而像 `reduce` 这样的聚合是一个动作操作，因为 `reduce` 需要（根据键）对 RDD 进行重新分区，并计算和返回某个聚合值，如和或平均值。Spark 中大多数动作的设计初衷都只是为了输出——返回单个值或小的值列表，或者将数据写回分布式存储。

Spark 的另一个好处是，它会“延迟”应用转换操作，即向集群提交作业以执行作业之前，检查完整的转换序列和一个动作。这种延迟执行机制带来了明显的存储和计算优化，因为它允许 Spark 建立数据转换过程（lineage）并评估完整的转换链，以便只计算结果所需的数据。例如，如果对 RDD 运行 `first()` 动作，Spark 将不会读取整个数据集，并只返回第一个匹配行。

4.1.3　使用RDD编程

Spark 应用程序的编写与之前在 Hadoop 上实现的其他数据流框架很像。代码被写在驱动程序（driver program）中，提交时在驱动程序所在机器上被延迟评估。一旦遇到动作，驱动程序代码就被分发到集群上，由 worker 节点在各自的 RDD 分区上执行该代码。然后结果被发送回驱动程序以进行聚合或汇编。如图 4-2 所示，驱动程序通过将来自 Hadoop 数据源的数据集并行化，创建一个或多个 RDD，应用操作来转换 RDD，然后对经过转换的 RDD 调用某个动作以检索输出。

> 术语并行化已经出现好几次了，有必要来解释一下。RDD 是分区的数据集合，允许程序员并行地对整个集合应用操作。正是分区支持了并行化，而且分区本身也是数据列表中计算的边界，其中数据存储在不同的节点上。因此，“并行化”是一种行为，它将数据集分区，并将数据的每个部分发送到将对其执行计算的节点。

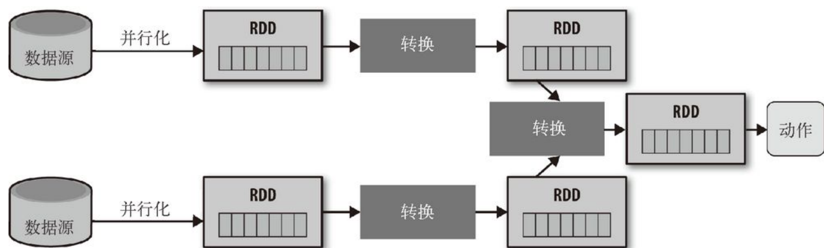


图 **4-2**：一个将集群中的数据集并行化（分区）为 **RDD** 的典型 **Spark** 应用程序

Spark 编程的典型数据流序列如下所示。

(1) 通过访问存储在磁盘（例如 HDFS、Cassandra、HBase 或 S3）上的数据、并行化某个集合、转换现有 RDD 或缓存来定义一个或多个 RDD。缓存是 Spark 的一个基本过程，即在节点内存中存储 RDD，用于在计算过程中快速访问。

(2) 通过将闭包（这里指不依赖于外部变量或数据的函数）传递到 RDD 的每个元素，来对 RDD 执行操作。Spark 提供了除 `map` 和 `reduce` 之外的许多高级算子。

(3) 对生成的 RDD 使用聚合动作（例如计数、收集、保存等）。动作将启动集群上的计算，因为在计算聚合之前无法进行任何计算。

简单解释一下变量和闭包，这是两个在 Spark 中容易混淆的概念。当 Spark 在 worker 节点上运行闭包时，闭包中使用的所有变量都将被复制到该节点，但在该闭包的局部范围内维护。如果需要外部数据，Spark 提供了两种类型的共享变量——广播变量和累加器，所有的 worker 节点都可以通过受限方式与它们交互。广播变量被分发给所有 worker 节点，但是只读的，并且通常作为查找表或禁用词列表使用。累加器是一个变量，worker 节点可以“累加”（满足结合律）它，通常用作计数器。这些数据结构类似于 MapReduce 中的分布式缓存和计数器，并且发挥着类似的作用。但是，由于 Spark 支持一般的进程间通信，所有这些数据结构可以用于更广泛的应用程序。

> 闭包是一种炫酷的函数式编程技术，使分布式计算得以实现。它们提供词法作用域名称绑定，这基本就意味着闭包是包括自己的独立数据环境的函数。由于存在这种独立性，闭包的运算不使用外部信息，因此是可并行化的。闭包在日常编程中越来越常见，通

常用作回调。在其他语言中，也可能被称为块或匿名函数。

尽管以下内容包含了如何使用 Spark 来执行分布式计算的示例，但 Spark 开发人员可用的转换和动作的完整指南不在本书的讨论范围之内。有关 Spark 所支持的转换和动作的完整列表以及使用文档，可以在 Spark Programming Guide（http://spark.apache.org/docs/latest/programming-guide.html）中找到。下一节将讨论如何使用 Spark 以交互方式在命令行上使用转换和动作，而免于编写完整的程序。

Spark 执行机制

以下内容是 Spark 执行机制的简要说明。Spark 应用程序本质上是独立运行的进程的集合，由驱动程序中的 SparkContext 进行协调。该上下文将与某个分配系统资源的集群管理器（例如 YARN）连接。集群中的每个 worker 节点都由一个 executor 管理，executor 又由 SparkContext 管理。executor 管理每台机器上的计算、存储和缓存。驱动程序、YARN 和 worker 节点的交互如图 4-3 所示。

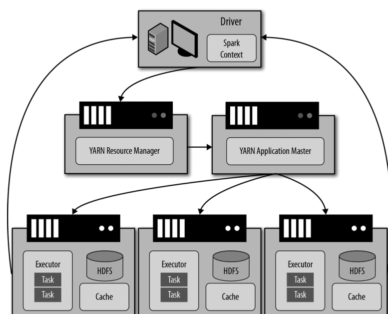


图 **4-3**：在 **Spark** 执行模型中，驱动程序是处理的一个重要部分

但要注意，应用程序代码从驱动程序发送到 executor，executor 指定上下文和要运行的各种任务。executor 与驱动程序来回通信以进行数据的共享或交互。驱动程序是 Spark 作业的关键参与者，因此它们应该和集群待在同一个网络上。这不同于 Hadoop 代码——你可以从任何地方将作业提交到

ResourceManager，并由 ResourceManager 负责作业在集群上的执行。

考虑到这一点，就能明白其实可以通过两种模式将 Spark 应用程序提交到 Hadoop 集群，分别是 yarn-client 和 yarn-cluster。在 yarn-client 模式下，驱动程序在客户端进程内运行。如上所述，ApplicationMaster 仅管理作业的进度并请求资源。然而在 yarn-cluster 模式下，驱动程序在 ApplicationMaster 进程内部运行，因此释放了客户端进程，像传统的 MapReduce 作业一样运行。如果程序员想获取即时结果或想以交互模式运行，可以使用 yarn-client 模式；而对于时间运行长或不需要用户干预的作业，使用 yarn-cluster 模式更为合适。

4.2 基于PySpark的交互性Spark

Spark 处理起可以放入集群内存的数据集非常快，足以支持数据科学家在实现了 Python REPL（read-evaluate-print loop，读取、评估、打印循环）的交互式 shell 中交互并探索大数据。Spark 中的交互式 shell 叫 pyspark。这种交互方式类似于在 Python 解释器中与本地 Python 代码交互、在命令行中编写命令并接收 stdout 的输出（还有 Scala 和 R 的交互式 shell）。这种类型的交互还支持交互式 notebook，在 Spark 环境中设置 iPython 或 Jupyter notebook 也非常容易。

本节将开始研究如何在 pyspark 中使用 RDD，因为这是启用 Spark 最简单的方法。为了运行交互式 shell，你需要定位 pyspark 命令，该命令位于 Spark 库的 bin 目录。和 $HADOOP_HOME（一个指向系统上 Hadoop 库的位置的环境变量）类似，你也应该配置一个 $SPARK_HOME。Spark 无须配置即可运行，因此只需下载适用于系统的 Spark 就足够了。将 $SPARK_HOME 替换为下载路径（或设置你的环境）就可以运行交互式 shell，如下所示：

```
hostname $ $SPARK_HOME/bin/pyspark
[... snip ...]
>>>
```

PySpark 使用本地 Spark 配置自动创建了一个 SparkContext。它

通过 sc 变量将自己暴露给终端。来创建第一个 RDD 吧：

```
>>> text = sc.textFile("shakespeare.txt")
>>> print text
shakespeare.txt MappedRDD[1] at textFile at NativeMethodAccessorImpl.java:
```

textFile 方法将莎士比亚全集（http://bit.ly/16c7kPV）从本地磁盘加载到名为 text 的 RDD 中。如果你检查 RDD，就会看到它是一个 MappedRDD，并且文件的路径是当前工作目录（你系统中要传入 shakespeare.txt 文件的正确路径）的相对路径。与第 2 章中的 MapReduce 示例类似，先来转换这个 RDD，以便计算分布式计算中的“Hello, World”，并使用 Spark 实现单词计数应用程序：

```
>>> from operator import add
>>> def tokenize(text):
...     return text.split()
...
>>> words = text.flatMap(tokenize)
```

我们导入了算子 add，一个可以用作加法闭包的命名函数，等会儿就要用到它。第一件要做的事是将文本拆分成单词：创建一个名为 tokenize 的函数（其参数是某个文本片段），通过简单地使用空格拆分文本返回该文本中的令牌（单词）列表。然后，通过应用 flatMap 算子并将 tokenize 闭包传递给它，转换 text RDD 以创建一个叫作 words 的新 RDD。

此时，我们有了一个类型为 PythonRDD 的 RDD words。你可能已经注意到，输入这些命令之后立即就产生了结果（尽管有一个意料之中的轻微处理延迟，因为要将莎士比亚全集拆分成单词）。因为 Spark 执行延迟评估，所以处理的执行（读取数据集、跨进程的分区，以及将 tokenize 函数映射到集合）还未发生。相反，PythonRDD 描述了创建此 RDD 的前提，并且在描述中还维护了数据到单词的转换过程。

因此，我们可以继续对这个 RDD 应用转换，免于经历漫长的分布式执行过程，而且还可能是个存在错误或非最佳的执行过程。如第 2 章所述，接下来的步骤是将每个单词映射到一个键值对，其中键是单词，值是 1，然后使用 reducer 对每个键的 1 进行求和。首先应用 map：

```
>>> wc = words.map(lambda x: (x,1))
>>> print wc.toDebugString()
(2) PythonRDD[3] at RDD at PythonRDD.scala:43
 | shakespeare.txt MappedRDD[1] at textFile at NativeMethodAccessorImpl.jav
 | shakespeare.txt HadoopRDD[0] at textFile at NativeMethodAccessorImpl.jav
```

这次不使用命名函数，而是使用匿名函数（在 Python 中使用 lambda 关键字）。这行代码将 lambda 映射到 words 中的每个元素。因此，每个 x 是一个单词，并且该单词将被匿名闭包转换成一个元组（单词,1)。为了检查到目前为止的转换过程（lineage），可以使用 toDebugString 方法来查看 PipelinedRDD 是如何转换的。然后应用 reduceByKey 动作获取单词计数，将它们写入磁盘：

```
>>> counts = wc.reduceByKey(add)
>>> counts.saveAsTextFile("wc")
```

一旦调用 saveAsTextFile 动作，分布式作业就会启动。当作业“在集群中”（或者仅作为本地机器上的多个进程）运行时，你会看到大量的 INFO 语句。如果退出解释器，就会在当前工作目录中看到一个名为 wc 的目录：

```
hostname $ ls wc/
_SUCCESS   part-00000 part-00001
```

每个 part 文件表示最终 RDD 的一个分区，最终 RDD 由计算机上的各种进程计算并保存到磁盘。如果对一个 part 文件使用 head 命令，则会看到单词计数对的元组：

```
hostname $ head wc/part-00000
(u'fawn', 14)
(u'Fame.', 1)
(u'Fame,', 2)
(u'kinghenryviii@7731', 1)
(u'othello@36737', 1)
(u'loveslabourslost@51678', 1)
(u'1kinghenryiv@54228', 1)
(u'troilusandcressida@83747', 1)
(u'fleeces', 1)
(u'midsummersnightsdream@71681', 1)
```

请注意，在 MapReduce 作业中，由于 map 和 reduce 之间有

shuffle 阶段和 sort 阶段，所以键会被排序。但因为所有 executor 都可以相互通信，所以 Spark 进行 reduce 时不会对重新分区排序。因此，前面的输出不会按照字母顺序排列。不过，由于用 `reduceByKey` 算子聚合了 `counts` RDD，所以即使没有排序，仍然能保证每个键在所有 part 文件中仅出现一次。如果需要排序，可以使用 `sort` 算子确保所有键在写入磁盘之前都已被排序。

4.3 编写Spark应用程序

使用 Python 编写 Spark 应用程序与在交互式控制台中使用 Spark 很像，因为 API 是相同的。但是你不需要在交互式 shell 中输入命令，而是需要创建一个完整的、可执行的驱动程序并将其提交到集群。这涉及一些在 `pyspark` 中自动处理的内务任务，包括获取 `SparkContext` 的访问，这是由 shell 自动加载的。

因此，许多 Spark 程序都是简单的 Python 脚本。它包含一些数据（共享变量），定义用于转换 RDD 的闭包，并描述 RDD 转换和聚合的分步执行计划。使用 Python 编写 Spark 应用程序的基本模板如下所示：

```
## Spark应用程序，使用spark-submit执行

## 导入
from pyspark import SparkConf, SparkContext

## 共享变量和数据
APP_NAME = "My Spark Application"

## 闭包函数
## 主要功能
def main(sc):
    """
    这里描述RDD转换和动作
    """
    pass

if __name__ == "__main__":
    # 配置Spark
    conf = SparkConf().setAppName(APP_NAME)
    conf = conf.setMaster("local[*]")
    sc   = SparkContext(conf=conf)

    # 执行主要功能
    main(sc)
```

此模板展示了 Python 语言的 Spark 应用程序自上而下的结构：import 让各种 Python 库以及 Spark 组件（例如 GraphX 和 SparkSQL）可用于分析。为了调试和日志记录，共享数据和变量被指定为模块常量，包括在 Web UI 中使用的应用程序名称。为了便于调试，可以让驱动程序包含特定于作业的闭包或自定义算子，这些闭包或自定义算子也可以导入到其他 Spark 作业中。最后，main 方法定义转换和聚合 RDD 的分析方法，该 main 方法作为驱动程序运行。

资深 Python 程序员应该会注意到这里使用了 if __name__ == '__main__'（通常被称为 ifmain）语句，其中 Spark 配置和 SparkContext 被定义并传递给 main 函数。通过 ifmain 可以轻松地将驱动程序代码导入到其他 Spark 上下文，无须创建新的上下文或配置、执行作业（在导入时，名称不是 __main__ ）。具体来说，Spark 程序员通常会将代码从应用程序导入到 iPython、Jupyter notebook 或 pyspark 交互式 shell 中，以便在对较大的数据集运行作业之前进行分析。

驱动程序定义了 Spark 执行过程的方方面面，例如程序员可以在代码中使用 sc.stop() 或 sys.exit(0) 停止或退出程序。这样的控制也可以扩展到执行环境——在这个模板中，Spark 集群的配置 local[*] 通过 setMaster 方法硬编码到 SparkConf 中，这告诉 Spark 在本地机器上运行尽可能多的进程（多进程，但不是分布式计算）。虽然你可以在命令行使用 spark-submit 来指定 Spark 执行的位置，但是驱动程序通常基于使用 os.environ 的环境变量来进行选择。因此，在开发 Spark 作业（例如使用 DEBUG 变量）时，作业可以在本地运行；但是在生产环境中，作业在集群的较大数据集上运行。

编写 Spark 应用程序肯定与编写 MapReduce 应用程序不同，因为转换和动作提供了灵活性，以及更灵活的编程环境。在下一节中，我们将看到一个完整的分析，它利用驱动程序的中心性来计算集群中的数据，从而创建一个可视化输出。

使用Spark可视化航班延误

第 3 章探讨了如何根据美国交通部的准点航班数据集（http://bit.ly/1Dz76xB），使用 Hadoop Streaming 和 MapReduce 计算每个机场的平均航班延误时间。这种解析 CSV 文件并执行聚合的计算是 Hadoop 的一个极其常见的用例，主要由于 CSV 数据很容易从关系数据库导出。该数据集记录了美国所有国内航班的起飞时间、到达时间

及其延误时间。有趣的是，虽然单月的数据很容易计算，但整个数据集因为数据量太大，所以必须使用分布式计算。

本例将使用 Spark 来执行数据集的聚合，具体来说，是确定哪些航空公司在 2014 年 4 月的总延误时间最长。由于 Spark 的 Python API 更加灵活，所以可以具体看看能使用哪些稍微高级些（和 Pythonic）的技术。此外，我们将通过 `matplotlib` 将结果拉取回来并在驱动程序机器上显示可视化图表，从而证明驱动程序对计算有多么重要（如果用传统的 MapReduce 执行此任务，则需要两个步骤）。

为了了解 Spark 应用程序的结构，并学习上一节中描述的模板的实际用法，我们将从宏观上把握程序的整体结构而忽略其细节：

```
## 导入
import csv
import matplotlib.pyplot as plt

from StringIO import StringIO
from datetime import datetime
from collections import namedtuple
from operator import add, itemgetter
from pyspark import SparkConf, SparkContext

## 模块常数
APP_NAME = "Flight Delay Analysis"
DATE_FMT = "%Y-%m-%d"
TIME_FMT = "%H%M"

fields   = ('date', 'airline', 'flightnum', 'origin', 'dest', 'dep',
            'dep_delay', 'arv', 'arv_delay', 'airtime', 'distance')
Flight   = namedtuple('Flight', fields)

## 闭包函数
def parse(row):
    """
    解析一行并返回一个命名元组
    """
    pass

def split(line):
    """
    使用csv模块分割行的函数
    """
    pass

def plot(delays):
    """
    显示航空公司总延误状况的柱状图
    """
```

```
    pass

## 主要功能
def main(sc):
    """
    描述数据集上应用的转换和动作，
    然后使用matplotlib绘制输出结果的可视化
    """
    pass

if __name__ == "__main__":
    # 配置Spark
    conf = SparkConf().setMaster("local[*]")
    conf = conf.setAppName(APP_NAME)
    sc   = SparkContext(conf=conf)

    # 执行主要功能
    main(sc)
```

这段代码虽然很长，但却是现实中 Spark 程序结构的一个良好概览。import 说明标准库工具和第三方库（如 matplotlib）通常混合使用。与 Hadoop Streaming 一样，任何不属于标准库的第三方代码都必须预先安装在集群上，或随作业一起提供。对于只需要在驱动程序上执行，而不需要在 executor 中执行的代码（如 matplotlib），可以使用 try/except 块捕获 ImportError。

> 与 Hadoop Streaming 一样，任何不属于 Python 标准库的第三方 Python 依赖都必须预先安装在集群中的每个节点上。但是，与 Hadoop Streaming 不同的是，Spark 有两个上下文（即驱动程序上下文和 executor 上下文），这允许一些重量级库（特别是可视化库）可以只安装在驱动程序机器上——只要它们不被在集群上运行的 Spark 运算使用的闭包使用就可以。如果想要预防错误，可以使用 try/except 块包裹 import 语句来捕获 ImportError。

然后，应用程序定义了一些可配置的数据，包括用于解析 datetime 字符串的日期和时间格式以及应用程序名称。它还创建了专用的 namedtuple 数据结构，以便从输入数据创建轻量级、可访问的行。此信息应该可为全部 executor 所用，但是它又足够轻量，所以不需要广播变量。接下来，定义处理函数 parse、split 和 plot，再使用 SparkContext 定义航空公司数据集上动作和转换的 main 函数。最后，ifmain 配置 Spark 并执行 main 函数。

有了这个完整的高级概览之后，再来深入了解一下代码的细节。从定义主要 Spark 操作和分析方法的 `main` 方法开始：

```
## 主要功能
def main(sc):
    """
    描述在数据集上应用的转换和动作，
    然后使用matplotlib绘制输出结果的可视化
    """

    # 加载航空公司查找字典
    airlines = dict(sc.textFile("ontime/airlines.csv").map(split).collect

    # 将查找字典广播到集群
    airline_lookup = sc.broadcast(airlines)

    # 读取CSV数据到一个RDD
    flights = sc.textFile("ontime/flights.csv").map(split).map(parse)

    # 映射总延误时间到航空公司（使用广播变量进行关联）
    delays = flights.map(lambda f: (airline_lookup.value[f.airline],
                                    add(f.dep_delay, f.arv_delay)))

    # 航空公司当月总延误时间
    delays = delays.reduceByKey(add).collect()
    delays = sorted(delays, key=itemgetter(1))

    # 驱动程序提供输出
    for d in delays:
        print "%0.0f minutes delayed\t%s" % (d[1], d[0])

    # 显示延误时间柱状图
    plot(delays)
```

第一个任务是从磁盘加载两个数据源：航空公司代码对应航空公司名称的查找表，以及航班数据集。数据集 airlines.csv 是一个小跳转表，允许通过航空公司代码获取完整的航空公司名称。不过由于这个数据集非常小，所以不必执行两个 RDD 的分布式连接，而是将此信息存储为 Python 字典，并使用 `sc.broadcast` 将其广播到集群中的每个节点。`sc.broadcast` 将本地 Python 字典转换为广播变量。

下面来说说这个广播变量的创建和执行。首先，从本地磁盘上的文本文件 airlines.csv（注意它的相对路径）创建一个 RDD。之所以需要创建 RDD，是因为这些数据可能来自 Hadoop 数据源，而该数据源可能由其位置的 URI（例如，HDFS 数据使用 hdfs://、S3 使用 s3:// 等）指定。请注意，如果这个文件只在本地机器上，则不需要将其加载到 RDD 中。然后，将 `split` 函数映射到数据集中的每个元素，

如上所述。最后，将 `collect` 动作应用于 RDD，它将数据作为 Python 列表从集群返回到驱动程序。因为应用了 `collect` 动作，所以当这行代码执行时，作业将被发送到集群以加载、拆分 RDD，并将上下文返回到驱动程序：

```
def split(line):
    """
    使用csv模块分割行的函数
    """
    reader = csv.reader(StringIO(line))
    return reader.next()
```

`split` 函数使用 `csv` 模块解析每行文本——使用 `StringIO` 创建一个带文本行的类似文件的对象，然后将其传递给 `csv.reader`。因为只有一行文本，所以可以简单地返回 `reader.next()`。虽然这种 CSV 解析方法看似相当重量级，但是它让我们能更轻松地处理分隔符、转义和 CSV 处理中的其他细枝末节。对于较大的数据集，通过类似的方法，使用 `sc.wholeTextFiles` 处理大量被分割为 128MB 的块（即 HDFS 上的块大小和副本大小）的整个 CSV 文件：

```
def parse(row):
    """
    解析行并返回一个命名元组
    """

    row[0]  = datetime.strptime(row[0], DATE_FMT).date()
    row[5]  = datetime.strptime(row[5], TIME_FMT).time()
    row[6]  = float(row[6])
    row[7]  = datetime.strptime(row[7], TIME_FMT).time()
    row[8]  = float(row[8])
    row[9]  = float(row[9])
    row[10] = float(row[10])
    return Flight(*row[:11])
```

接下来，`main` 函数加载更大的 flights.csv，后者需要使用 RDD 以并行方式计算。分割 CSV 行之后，将 `parse` 函数映射到 CSV 行，这可以将日期和时间转换为 Python 的日期和时间，并适当地转换浮点数。此函数的输出是一个名为 `Flight` 的 `namedtuple`，在应用程序的模块常量部分定义。命名元组是包含记录信息的轻量级数据结构，允许通过名称（例如 `flight.date`）而不是位置（例如 `flight[0]`）来访问数据。和普通的 Python 元组一样，命名元组也不可变，因此可以放心地将它们用于处理程序，数据不会被修改。此外，与字典相比，它们占用的内存更小，处理效率也更高。因此，命

名元组在内存尤显珍贵的大数据应用程序（如 Spark）中具有显著的优势。

有了 `Flight` 对象的 RDD 后，我们要做的最后一个转换是映射一个匿名函数，将 RDD 转换成一系列键值对，其中键是航空公司的名称，值是到达和起飞的延误时间之和。目前除了创建航空公司字典之外，还没有对该集合执行过任何操作。不过一旦开始使用 `reduceByKey` 动作和 `add` 算子计算每个航空公司的延误时间之和，该作业就将在集群中执行，运算结果将被收集回驱动程序。

此时，集群计算已经完成，驱动程序以串行方式继续运行。延误时间按照延误程度在客户端程序的内存中进行排序。请注意，之所以能这么做，原因和创建航空公司查找表作为广播变量的原因相同：航空公司的数量很少，在内存中排序的效率更高。但如果这个 RDD 非常大，就可以使用 `rdd.sort` 进行分布式排序。最后，我们没有将结果写入磁盘，而是将输出打印到控制台。如果数据集较大，则可以使用 `rdd.first` 动作来获取前 `n` 个项目，而不用打印整个数据集；或者可以使用 `rdd.saveAsTextFile` 将数据写回到本地磁盘或 HDFS。

最后，因为数据在驱动程序中可用，所以可以使用 `matplotlib` 可视化结果，如下所示：

```
def plot(delays):
    """
    显示航空公司总延误状况柱状图
    """
    airlines = [d[0] for d in delays]
    minutes  = [d[1] for d in delays]
    index    = list(xrange(len(airlines)))

    fig, axe = plt.subplots()
    bars = axe.barh(index, minutes)

    # 在右侧添加总分钟数
    for idx, air, min in zip(index, airlines, minutes):
        if min > 0:
            bars[idx].set_color('#d9230f')
            axe.annotate(" %0.0f min" % min, xy=(min+1, idx+0.5), va='cent
        else:
            bars[idx].set_color('#469408')
            axe.annotate(" %0.0f min" % min, xy=(10, idx+0.5), va='center'

    # 设置tick
    ticks = plt.yticks([idx+ 0.5 for idx in index], airlines)
    xt = plt.xticks()[0]
```

```
    plt.xticks(xt, [' '] * len(xt))

    # 最小化图表垃圾
    plt.grid(axis = 'x', color ='white', linestyle='-')

    plt.title('Total Minutes Delayed per Airline')
    plt.show()
```

希望此示例能说明集群和驱动程序的交互过程（发送数据进行分析，然后将结果返回到驱动程序），以及 Python 代码在 Spark 应用程序中起到的作用。要运行此代码（假定你有一个名为 ontime 的目录，其中有这两个 CSV 文件），请使用 `spark-submit` 命令，如下所示：

```
hostname $ spark-submit app.py
```

因为在 `ifmain` 中将 master 节点的配置硬编码为了 `localhost[*]`，所以此命令创建了一个 Spark 作业，它有 localhost 上尽可能多的进程。这个 Spark 作业将使用本地 `SparkContext` 执行 `main` 函数中指定的转换和动作：首先，将跳转表加载为 RDD，收集它并将它广播给所有进程；然后，加载航班数据 RDD 并以并行方式计算平均延误时间。

一旦上下文和 `collect` 的输出返回到驱动程序，就可以使用 `matplotlib` 来可视化结果了，如图 4-4 所示。最终结果显示，2014 年 4 月，夏威夷航空公司和阿拉斯加航空公司的总延误时间（以分钟为单位）最短，甚至有提前达到，其余大航空公司均有延误。本例的新颖之处不在于分析的可视化，而是在一步步提交并行执行作业的过程中，在合理的时间内显示结果。因此，这类直接向用户提供按需分析以立即获得有用信息的应用程序变得越来越普遍。

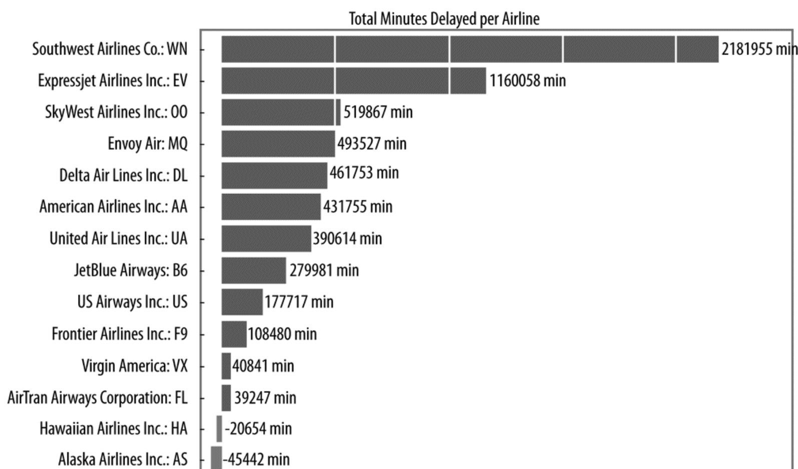


图 **4-4**：数据集中延误时间最长的航空公司的可视化图

4.4　小结

虽然 Spark 最初旨在突破 MapReduce 在执行迭代算法方面的局限性，但它现在已经发展成为了一个完整的通用分布式计算引擎。Spark 已经应用于大量大数据处理工作负载——它们都使用了 Spark 这一通用引擎，而没有实现专用系统。因为 Spark 比传统的 MapReduce 快 10~20 倍，所以你可能想知道 Spark 在 Hadoop 生态系统中处于什么位置。虽然说 Spark 终将替代 MapReduce 还为时过早，但它已经吸引了相当多采用了 Hadoop 的企业和公司，这些组织需要一个可以使用现有的集群资源执行接近实时计算的平台。但是要记住，至少现在 Spark 还不是 Hadoop 和 MapReduce 的替代品，而应该被认为是它们的扩展，并且可以与 Hadoop 生态系统的其他组成部分并存。

Spark 不解决分布式存储问题（通常 Spark 从 HDFS 获取其数据），但它为分布式计算提供了丰富的函数式编程 API。这种框架建立在弹性分布式数据集（RDD）的理念上。RDD 是一种编程抽象，表示分区的对象集合，允许对它们执行分布式操作。RDD 是容错的（弹性部分），还可以存储在 worker 节点的内存中，以便快速重用。内存存储提供更快、更容易表达的迭代算法，还支持实时交互式分析。

因为 Spark 库拥有 Python、R、Scala 和 Java 的 API，以及用于机器学习、流式数据、图形算法和类 SQL 查询的内置模块，所以它已经迅速成为现今最重要的分布式计算框架之一。

当与 YARN 结合时，Spark 用于增强（不是替换）现有的 Hadoop 集群。未来，它将成为大数据的重要组成部分，开辟数据科学探索的新途径。

这一章远不止是 Spark 的完整介绍；相反，它介绍了 Spark 计算框架和弹性分布式数据集，提供了如何与 Spark 交互和编程的思路。因为本书的目标读者是了解 Python 或 R 的数据科学家，所以他们可能觉得 Spark 比 Hadoop Streaming 更适合作分析。在本书余下的内容中，我们将使用 Hadoop Streaming 和 Spark 进行计算，但在大多数情况下——特别是与机器学习相关时——将主要使用 Spark。

第 5 章　分布式分析和模式

MapReduce 和 Spark 让开发人员和数据科学家能轻松进行数据并行运算。在这类运算中，数据被分发到多个处理节点同时计算，然后通过 reduce 产生最终输出。而 YARN 提供了简单的任务并行性，通过为每个任务分配自由计算资源，支持集群同时执行多个不同的操作。并行缩短了执行单个计算所需的时间，从而支持以每秒数千条记录的速度分析 PB 级的数据集，或处理由多个异构数据源组成的数据集。然而，大多数并行运算（如前述运算）都比较简单。这也带来了一个问题：数据科学家如何进行大规模高级数据分析？

不妨用 Creighton Abrams 的一句话总结一下大规模分析的主要原则：“要想吃掉一头大象，也得一口一口来。”单个运算只对数据进行多个微小的处理，而要想得到更有意义的结果，必须将这些运算组成一个被称为数据流的分步序列。如果两个运算可以同时进行，则数据流可以分叉（fork）和合并（merge），支持任务和数据并行，但是必须保证序列的数据从输入数据源串行馈送到最终输出。因此，数据流被描述为有向无环图（DAG）。如果一种算法、分析或精心设计的计算可以表示为 DAG，则它可以在 Hadoop 上并行化，认识到这一点非常重要。

不幸的是，你很快就会发现，许多算法都不能轻易转换为 DAG，也就不适合这种类型的并行化了。不能被描述为有向数据流的算法有：在整个计算过程中维持或更新单个数据结构的算法（需要一些共享内存），或者依赖中间步骤计算结果的算法（需要中间进程间通信）。引入循环的算法，特别是循环次数不限的迭代算法，也不容易描述为

DAG。

有一些工具和技术可以满足 MapReduce 和 Spark 中循环性、共享内存或进程间通信的需求。但是要利用这些工具，必须将算法重写为分布式形式。比起重写算法，更常采用的是一种技术要求更低但同样有效的方法：设计一种数据流，将输入域分解为适合单个机器内存的较小输出，对输出运行串行算法，然后使用另一个数据流在集群上验证该分析（例如计算误差）。

正是因为这种方法被广泛采用，Hadoop 才被普遍认为是一个释放大数据集潜力的预处理器——它通过每个操作将数据集规约（reduce）成越来越容易管理的块。一种常见做法是，使用 MapReduce 或 Spark 将数据分解到一个可以载入 128GB 内存（高性价比机器的硬件要求）的计算空间中。这个规则通常被称为“最后一英里”（last-mile）计算，因为它将数据从极大的空间移动到足够近的地方（即最后一英里），从而能够进行准确的分析或特定于应用程序的计算。

在本章中，我们将在数据流的上下文中探讨将计算空间缩小或分解为更易于管理的计算空间的并行计算模式。首先，讨论基于键的计算，这是 MapReduce 的需求，对 Spark 也至关重要；接着，学习概要（summarization）、索引（indexing）和过滤（filtering）的模式，这些模式是大多数分解算法的关键部分——在这个上下文中，我们将讨论统计概要、抽样、搜索和分类（binning）的应用；最后，纵览三种回归、分类和聚类风格分析的预处理技术。

> 本章将介绍一些 Hadoop 生态系统中使用的方法。这些方法也被其他项目使用，具体内容将在最后 4 章讨论。本章还将讨论表示为数据流的算法，而第 8 章将接着讨论用于创建数据流的工具，包括高级 API（如 Pig 和 Spark Data Frames）。本章讨论的许多过滤和概要算法更容易表示为结构化查询，第 7 章将具体讨论如何在 Hadoop 上使用 Hive 执行结构化查询。最后，这些章节中的组件（包括 Sckit-Learn 模型的用法）是理解使用 Spark 的 MLlib 进行机器学习的第一步，这将在第 9 章中讨论。

本章还介绍了经常用于数据分析的标准算法，包括统计概要（并行版的“describe”命令）、并行 grep、TF-IDF 和 canopy 聚类。通过这些例子，我们将阐明 MapReduce 和 Spark 的基本机制。

5.1 键计算

要理解数据流实际是如何工作的，第一步就是理解键值对和并行计算之间的关系。在 MapReduce 中，所有数据在 map 阶段和 reduce 阶段都被构造为键值对。关键需求主要与 reduce 有关，因为聚合是按键分组的，并行 reduce 需要对键空间（换句话说，就是键的值域）进行分区，使每一个 reducer 任务都能收到一个键的所有值。如果没有用于分组的键（这其实很常见），你就可以按单一的键进行 reduce，强制对所有映射值进行 reduce。然而在这种情况下，reduce 阶段无法从并行中受益。

键虽然经常被忽略（特别是在 mapper 中，键仅仅是文档标识符），但是它能让计算在数据集上同时进行。因此，数据流表达了一组值与另一组值之间的关系。这听起来很耳熟，尤其是在更传统的数据管理方式的上下文中，它相当于关系数据库的结构化查询。就像你不会在 PostgreSQL 这样的数据库上运行多个单独的查询来进行不同维度的分析，MapReduce 和 Spark 计算采用了并行执行分组操作，如图 5-1 所示的按键分组的平均值计算。

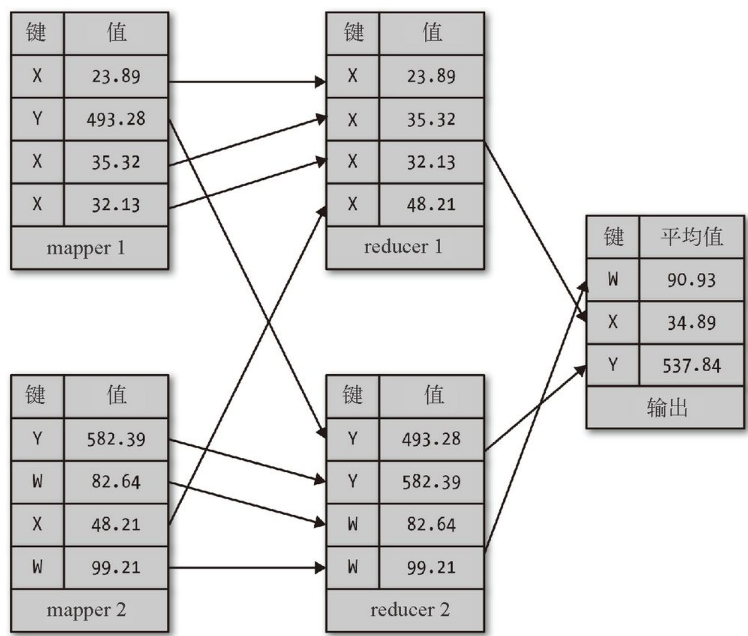


图 **5-1**：通过将键空间分区到多个 **reducer**，键支持并行 **reduce**

此外，键可以保存在数据流的一个阶段中已经被 reduce 的信息，还能自动并行下一步计算所需的结果。这是通过复合键完成的，下一节将讨论这种技术，它表明键不一定是简单的、原始的值。事实上，键对于这些类型的计算非常有用，尽管它们在使用 Spark 的计算中不是必需品（RDD 可以是简单值的集合），但大多数 Spark 应用程序需要它们来进行分析，主要是使用 `groupByKey`、`aggregateByKey`、`sortByKey` 和 `reduceByKey` 动作来收集和 reduce。

5.1.1　复合键

键不一定是简单的原始数据类型，如整型或字符串；相反，它们可以是复合类型或复杂类型，只要是可散列（hashable）且可比较（comparable）的即可。可比较的类型必须至少能暴露某种用于判断相等的机制（用于 shuffle）和某种排序方法（用于 sort）。一般通过将某种类型映射到一个数值（例如，将月份映射到整数 1~12），或通过词汇顺序来完成比较。Python 中的可散列类型是任意一种不可变类型，最典型的就是元组。但是元组可以包含可变类型（例如一个列表的元组），因此可散列的元组是指由不可变类型组成的元组。像列表和字典这样的可变类型可以转换为不可变的元组。

```
# 将列表转换为元组
key = tuple(['a', 'b', 'c'])

# 将字典转换为由元组构成的元组
key = {'a': 1, 'b': 2}
key = tuple(key.items())
```

使用复合键的方式主要有两种：在多个维度上划分键空间，以及在仅涉及值的计算阶段携带特定于键的信息。思考以下形式的 Web 日志记录：

```
local - - [30/Apr/1995:21:18:07 -0600] "GET 7448.html HTTP/1.0" 404 -
local - - [30/Apr/1995:21:18:42 -0600] "GET 7448.html HTTP/1.0" 200 980
remote - - [30/Apr/1995:21:22:56 -0600] "GET 4115.html HTTP/1.0" 200 1363
remote - - [30/Apr/1995:21:26:29 -0600] "GET index.html HTTP/1.0" 200 2881
```

Web 日志记录是 Hadoop 大数据计算的典型数据源，因为它们代表每个用户的点击流式数据，从中可以轻松探究数据的多个方面。此外，它们往往也是非常大的动态半结构化数据集，非常适合在 Spark 和 MapReduce 中进行运算。要对该数据集进行初始计算，则先需要进

行频率分析。例如，可以使用复合键将文本分解为两个每日时间序列，一个用于本地流量，另一个用于远程流量。

```
import re
from datetime import datetime

# 解析日志记录中的日期时间
dtfmt = "%d/%b/%Y:%H:%M:%S %z"

# 使用正则表达式解析日志记录
linre = re.compile(r'^(\w+) \- \- \[(.+)\] "(.+)" (\d+) ([\d\-]+)$')

def parse(line):
    # 使用正则表达式匹配日志记录
    match = linre.match(line)
    if match is not None:
        # 正则表达式含有分组，能提取日志源、时间戳、
        # 请求、状态码和响应字节大小
        parts = match.groups()

        # 解析日期时间，返回日志源、年和天
        date = datetime.strptime(parts[1], dtfmt).timetuple()
        return (parts[0], date.tm_year, date.tm_yday)
```

将此函数用于 mapper 可以解析日志文件的每一行，或作为闭包传递给从文本文件中加载的 RDD 的 `map` 方法。`parse` 函数使用一个日期格式和一个正则表达式来解析行文本，然后发出由流量类型、年份和当天在当年中的位置组成的复合键。该键与计数器（例如数字 1）相关联，计数器可以被传递到求和 reducer 中以获取基于频率的时间序列。映射该数据集生成了以下数据：

```
('local', 1995, 120)   1
('local', 1995, 120)   1
('remote', 1995, 120)   1
('remote', 1995, 120)   1
```

用作复杂键的复合键支持在键空间的多个面上进行计算，例如网络流量的来源、年份和天，这是复合键最常见的用例。另一个常见用例是将特定键的信息传播到下游计算，例如依赖于 reduce 或每个键的聚合值的计算。通过将 reducer 的计算与键（特别是类似计数或浮点数的值）相关联，这些信息能与键一起被维护，以用于更复杂的计算。

 MapReduce、Spark 的 Java 以及 Scala API 都需要使用强类型的键和值。就 Hadoop 而言，这意味着

> 复合键和结构化的值需要被定义为实现 `Writable` 接口的类，并且键必须实现 `WritableComparable` 接口。这些工具使 Java 和 Scala 开发人员能够使用轻量级且可扩展的数据结构序列化功能，从而最大限度上减少了网络流量并支持 shuffle 和 sort 操作。然而，Python 开发人员需要自己将元组和 Python 原始类型序列化成字符串，还得将它们转换回来。要想序列化嵌套数据结构，可以使用 `json` 模块。在处理更复杂的作业时，二进制序列化格式（如 Protocol Buffers、Avro 或 Parquet）可通过最小化网络流量来缩短处理时间。

复合数据序列化

使用复合键（和复杂的值）的最后一步是理解复合数据的序列化和反序列化。序列化是指将内存中的对象转换成字节流，使其可以被写入磁盘或通过网络传输（反序列化是指相反的过程）。序列化过程必不可少，特别是在 MapReduce 中，因为键和值在 map 阶段和 reduce 阶段之间被写入磁盘（通常作为字符串写入）。然而，理解 Spark 中的序列化也非常重要——Spark 的中间作业要对数据进行预处理，供后续计算使用。

在 Spark 中，Python API 默认使用 `pickle` 模块进行序列化，这意味着你使用的任何数据结构都必须是可以 pickle 的。虽然 `pickle` 模块非常高效，但在 Spark 编程中，这个约束可能沦为一个陷阱（gotcha），特别是传递闭包（不依赖于全局值的函数，通常是匿名 lambda 表达式）时。使用 MapReduce Streaming 时，必须将键和值序列化为字符串，并以指定的字符分隔，默认情况下为制表符（`\t`）。新的问题来了：有没有更高效的办法，能将复合键（和值）序列化为字符串呢？

我们通常会简单地使用内置的 `str` 函数对不可变类型（例如元组）进行序列化，将该元组转换为可以轻松 pickle 或流式传输的字符串。然后问题转向反序列化——通过 Python 标准库中的 `ast`（抽象语法树）模块，使用 `literal_eval` 函数评估元组字符串得到 Python 元组类型，如下所示：

```
import ast

def map(key, val):
    # 解析复合键，它是一个元组
    key = ast.literal_eval(key)
```

```
    # 以字符串写新的键
    return (str(key), val)
```

随着键和值越来越复杂，你也得考虑使用其他序列化数据结构了，尤其是那种更紧凑的、能减少网络流量或能转换为字符串值以确保安全性的数据结构。例如，结构化数据的常见表示形式是 Base64 编码的 JSON 字符串，因为它很紧凑，仅使用 ASCII 字符，并且很容易用标准库进行序列化和反序列化，如下所示：

```
import json
import base64

def serialize(data):
    """
    返回数据（键或值）的Base64编码的JSON表示
    """
    return base64.b64encode(json.dumps(data))

def deserialize(data):
    """
    解码Base64编码的JSON数据
    """
    return json.loads(base64.b64decode(data))
```

但使用复杂的序列化表示时要小心，通常需要权衡序列化的计算复杂度与它所需的空间。许多类型的并行算法更适合使用元组字符串或制表符分隔格式，实现起来更快、更简单，如果能管理好键在计算中的传递过程更能事半功倍。在下一节中，我们将看到 MapReduce 和 Spark 中常见的基于键的计算模式。

5.1.2　键空间模式

可以使用键计算的概念管理数据集及其关系。然而，键也是计算的主要部分。因此，除了数据之外，键也必须被管理。本节将探讨影响键空间的几种模式，特别是爆炸（explode）、过滤、变换和恒等（identity）模式。这些常见模式通过键和值之间的关联关系来构造更大的模式并完成算法。

以下示例的主角是一个订单数据集，其中键是订单 ID、客户 ID 和时间戳，值是订单所购买产品的商品统一代码（universal product code，UPC）列表，如下所示：

```
1001, 1063457, 2014-09-16 12:23:33, 098668259830, 098668318865
1002, 0171488, 2014-12-11 03:05:03, 098668318865
1003, 1022739, 2015-01-03 13:01:54, 098668275427, 098668331789, 0986682743
```

01. 键空间变换

最常见的基于键的运算是输入键的域的变换，在 `map` 和 `reduce` 中均可进行。在 map 期间变换键空间会导致数据在聚合期间重新分区（划分），而在 reduce 期间变换键空间可用于重组输出（或后续计算的输入）。最常见的变换函数是直接赋值、复合、分割和键值换位。

直接赋值丢弃了输入的键（通常被完全忽略），从输入值或别的来源（例如随机的键）构造新键。思考一下从文本、CSV 或 JSON 加载原始或半结构化数据的情况。在这种情况下，输入键是行或文档 ID，通常因为某些特定于数据的值而被丢弃。

如上一节所述，复合及其相反运算分割管理复合键。复合构建复合键，或向复合键添加新的元素，能增加键关系的面；分割拆分复合键，而只使用其中一小部分。通常，值也能被复合和分割，复合键从拆分值接收新的数据（反之亦然），以确保没有数据丢失。此外，也可以通过复合或分割，丢弃不需要的数据或删除无关的信息。

键值换位交换键和值，是一种常见模式，特别是在链式的 MapReduce 作业或依赖于中间聚合（特别是 `groupby`）的 Spark 操作中。例如，为了通过值而不是键对数据集进行排序，必须先在 map 中将键和值换位，执行 `sortByKey` 或者利用 MapReduce 中的 shuffle 和 sort，然后在 reduce 或另一个 map 中重新换位。

来看一个按照每个订单中的产品数量和日期为订单排序的作业，这将使用之前学过的所有键空间变换方法：

```
# 将订单加载到一个RDD，解析CSV
orders = sc.textFile("orders.csv").map(split)

# 键分配：(orderid, customerid, date), products
orders = orders.map(lambda r: ((r[0], r[1], r[2]), r[3:]))

# 计算订单大小，并将键拆分为orderid和date
orders = orders.map(lambda (k, v): ((k[0], parse_date(k[2])), len(v)
```

```
# 交换键和值，排序
orders = orders.map(lambda (k, v): ((v, k[1]), k[0]))

# 根据键将订单排序
orders = orders.sortByKey(ascending=False)

# 再次交换键和值，以便再次使用订单ID作为键
orders = orders.map(lambda (k,v ): (v, k))

# 根据订单大小和日期，获取前10个订单ID
print orders.take(10)
```

这个例子对于所需完成的任务可能有点冗长，但它确实演示了如下每种类型的变换。

(1) 如第 4 章讨论的，首先使用 `split` 方法从一个 CSV 文件中加载数据集。

(2) 此时的 `orders` 仅仅是一个列表集合，因此将值分解为 ID 和日期，将它们分配为键，将产品列表作为值，关联键与值。

(3) 下一步是获取产品列表的长度（订购的产品数量），并使用一个包装了 `datetime.strptime` 日期格式的闭包来解析日期。请注意，此方法拆分了复合键并删除了客户 ID，这其实没有必要。

(4) 为了将订单按大小排序，需要将订单大小的值和键换位，还要将日期从键中分割出来，以便也可以按日期排序。

(5) 执行排序后，键值重新换位，以便可以查看每个订单的大小和日期。

以下片段演示了第一条记录在经过 Spark 作业中的每个 map 时发生的变换：

```
0. "1001, 1063457, 2014-09-16 12:23:33, 098668259830, 098668318865"
1. [1001, 1063457, 2014-09-16 12:23:33, 098668259830, 098668318865]
2. ((1001, 1063457, 2014-09-16 12:23:33), [098668259830, 0986683188
3. ((1001, datetime(2014, 9, 16, 12, 23, 33), 2)
4. ((2, datetime(2014, 9, 16, 12, 23, 33)), 1001)
5. (1001, (2, datetime(2014, 9, 16, 12, 23, 33)))
```

经过这一系列变换，客户端程序就可以按订单大小和日期获取

前 10 个订单，并在分布式计算后将它们打印出来。

02. 爆炸**mapper**

爆炸 mapper 针对单个输入键生成多个中间键值对。一般来说，这是通过结合键移位（key shift）和将值拆分为多个部分来实现的。正如第 2 章中的单词计数示例，它根据空格拆分行，将输入 mapper 中的单个行号 / 行对输出为几个新的键值对，即单词 /1。通过将值按组成部分划分并且重新将键分配给它们，爆炸 mapper 还可以生成许多中间对。

在此示例中，可以按订单展开产品列表，得到订单 / 产品对，如下面的代码所示：

```
def order_pairs(key, products):
    # 返回订单id/产品对的列表
    pairs = []

    for product in products:
        pairs.append((key[0], product))

    return pairs

orders = orders.flatMap(order_pairs)
```

将这个 mapper 应用到输入数据集，会产生如下输出：

```
1001, 098668259830
1001, 098668318865
1002, 098668318865
1003, 098668275427
1003, 098668331789
1003, 098668274321
```

注意在 RDD 上使用的 `flatMap` 操作，这是专为爆炸 map 设计的。它的操作与常规 map 相似，但该函数产生一个序列而不是单个项，然后该序列被链接成单个集合（而不是列表的 RDD）。在 MapReduce 和 Hadoop Streaming 中不存在这样的限制，一个 `map` 函数可以发射任意数量的对（或者根本不发送）。

03. 过滤器**mapper**

尽管本章后面会更详细地讨论过滤（特别是统计抽样），但鉴于它与键的操作相关，此处也得先说两句。过滤通常对限制 reduce 阶段执行的计算量至关重要，特别是在大数据环境中。它还可将同一数据流的计算划分为两条路径，这是专为超大数据集设计的一种大型算法，是面向数据的分支方法。想想看在只选择了 2014 年订单的情况下，如何扩展订单示例（使用 Spark）：

```
from functools import partial

def year_filter(item, year=None):
    key, val = item
    if parse_date(key[2]).year == year:
        return True
    return False

orders = orders.filter(partial(year_filter, year=2014))
```

Spark 提供了一个过滤操作，它接受一个函数并转换 RDD，只保留函数返回 `True` 的元素。此示例展示了闭包的高级用法以及可以获取任何年份的通用过滤器函数。`partial` 函数创建一个闭包，其 `year_filter` 的参数 `year` 始终为 2014，支持更强大的功能。MapReduce 代码类似，但需要更多的逻辑：

```
def YearFilterMapper(Mapper):

    def __init__(self, year, **kwargs):
        super(YearFilterMapper, self).__init__(**kwargs)
        self.year = year

    def map(self):
        for key, value in self:
            if parse_date(key[2]).year == self.year:
                self.emit(key, value)

if __name__ == "__main__":
    mapper = YearFilterMapper(2014)
    mapper.map()
```

mapper 什么也不发射也完全没问题。因此，用于过滤器 mapper 的逻辑是仅在满足条件时才发射。通过使用基于类的 `Mapper`，并用想要过滤的年份简单地实例化类，可以获得与 `partial` 方法相同的灵活性。更高级的 Spark 和 MapReduce 应用程序可能从运行作业的命令上接收年份作为输入。

最后一个订单记录（订单 1003）被删除了，因为它是 2015 年的订单，过滤产生的其他数据与输入相同：

```
1001, 1063457, 2014-09-16 12:23:33, 098668259830, 098668318865
1002, 0171488, 2014-12-11 03:05:03, 098668318865
```

04. 恒等模式

MapReduce 中的最后一个常用键空间模式（一般不用于 Spark 中）是恒等（identity）函数。它只是一个传递，能使恒等 mapper 或者恒等 reducer 返回与输入相同的值（就好像在恒等函数 `f(x) = x` 中一样）。恒等 mapper 通常用于在数据流中执行多个 reduce。当在 MapReduce 中使用恒等 reducer 时，该作业等同于在键空间上进行排序。恒等 mapper 和恒等 reducer 的简单实现如下所示：

```
class IdentityMapper(Mapper):

    def map(self):
        for key, value in self:
            self.emit(key, value)

class IdentityReducer(Reducer):

    def reduce(self):
        for key, values in self:
            for value in values:
                self.emit(key, value)
```

因为 MapReduce 使用了最优的 shuffle 和 sort，因而恒等 reducer 通常更常见一些。不过恒等 mapper 也非常重要，特别是在链式 MapReduce 作业中，一个 reducer 的输出必须立即被第二个 reducer 再次 reduce。事实上，正是因为 MapReduce 的操作是分阶段的，所以才需要恒等 reducer。在 Spark 中，因为 RDD 被延迟评估，所以不需要恒等闭包。

5.1.3　pair与stripe

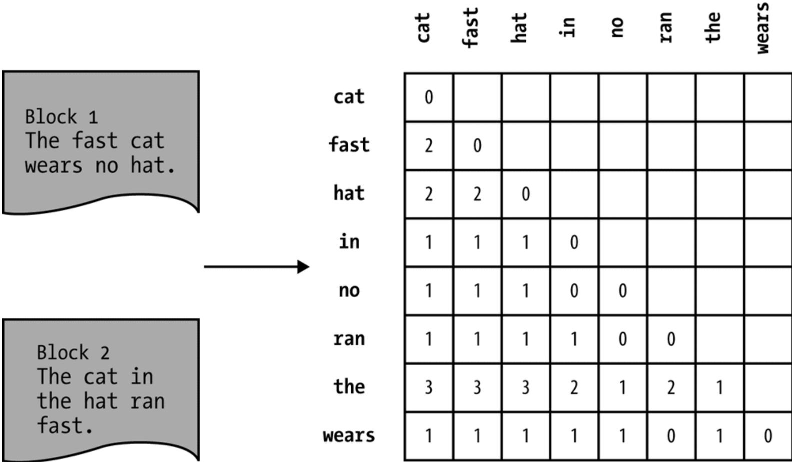
数据科学家习惯用表示为向量、矩阵或数据框的数据。线性代数计算往往针对单核机器进行了优化，而机器学习中的算法使用低级数据结构（如 `numpy` 库中的多维数组）来实现。这些结构虽然紧凑，但因为数据的量级实在太大，所以无法在大数据环境中使用。相反，矩阵

通常有两种表示方式：**pair** 和 **stripe**。pair 和 stripe 都是基于键的计算。

为了理解这一点，试着为基于文本的语料库建立单词共现矩阵（和单词计数一样，这是演示 pair 和 stripe 的典型问题。1）使用单词共现创建的语言的统计模型可用于机器翻译、句子生成等大量应用。

> 1Jimmy Lin, Chris Dyer, *Data-Intensive Text Processing with MapReduce* (http://bit.ly/1PcgEmB).

如图 5-2 所示的单词共现矩阵是大小为 N×N 的矩阵，其中 N 是语料库的词汇量（单词的种数）。每个单元 Wi, j 包含词 w_i 和词 w_j 同时出现在句子、段落、文档或其他固定长度窗口中的次数。这个矩阵很稀疏，特别是采用了积极的停用词过滤之后，因为大多数单词通常仅与非常少的其他词共现。



	cat	fast	hat	in	no	ran	the	wears
cat	0							
fast	2	0						
hat	2	2	0					
in	1	1	1	0				
no	1	1	1	0	0			
ran	1	1	1	1	0	0		
the	3	3	3	2	1	2	1	
wears	1	1	1	1	1	0	1	0

图 **5-2**：演示同一文本块（例如句子）中词条同时出现频率的单词共现矩阵

pair 方法将矩阵中的每个单元映射到特定值，其中词对是复合键 `i，j`。因此，reducer 对每个单元的值进行处理，以产生最终的单元挨单元的矩阵。这是一种合理的方法，它产生的输出中的每个 Wi, j 被单独计算并存储。使用求和 reducer，mapper 如下所示（有关使用 NLTK 进行文本处理和分词的更多信息，请参见第 3 章）：

```
from itertools import combinations
```

```
class WordPairsMapper(Mapper):

    def map(self):
        for docid, document in self:
            tokens = list(self.tokenize(document))
            for pair in combinations(sorted(tokens), 2):
                self.emit(pair, 1)
```

这个方法的重中之重是使用内置的 `sorted` 函数对令牌进行字典排序。在对称矩阵（$W_{i,j}$ 等于 $W_{j,i}$ 的矩阵）中，必须为词对排序，否则键 `(b,a)` 和 `(a,b)` 不会 reduce 在一起。请注意，`itertools` 库中的 `combinations` 函数保持其输入列表的排序。输入如下所示：

```
"See Spot run, run Spot, run!"
```

单词共现矩阵的词对聚合结果如下所示：

```
(run, run), 6
(run, see), 3
(run, spot), 6
(see, run), 3
(see, spot), 2
(spot, run), 6
(spot, see), 2
(spot, spot), 1
```

虽然 pair 方法易于理解和实现，但是它产生了许多中间对。这些中间对必须在网络上传输，这一过程在 MapReduce 的 shuffle 阶段和 sort 阶段，以及 `groupByKey` 操作将值 shuffle 到 RDD 各分区的期间均有发生。此外，pair 方法不太适用于需要整行（或列）数据的计算。

stripe 方法最初被设想为一种减少中间对的数量和网络通信的优化手段，从而让作业运行得更快。不过它很快也成为一种重要的工具，应用于许多需要针对一个元素执行快速计算（例如相对频率或其他统计运算）的算法中。stripe 方法没有使用词对，而是在 mapper 中为每个条目构造关联数组（Python 字典），并作为值发射：

```
from collections import Counter

class WordStripeMapper(Mapper):

    def map(self):
```

```
        for docid, document in self:
            tokens = list(self.tokenize(document))

            for i, term in enumerate(tokens):
                # 为每个条目创建新的stripe
                stripe = Counter()

                for j, token in enumerate(tokens):
                    # 不计算该条目与本身的共现
                    if i != j:
                        stripe[token] += 1

                # 发射条目和stripe
                self.emit(term, stripe)

class StripeSumReducer(Reducer):

    def reduce(self):
        for key, values in self:
            stripe = Counter()

            # 将所有计数器相加
            for value in values:
                for token, count in value.iteritems():
                    # 为每一个令牌分别累加stripe
                    stripe[token] += count

            self.emit(key, stripe)
```

stripe 的 mapper 和 reducer 有点复杂。在 mapper 中需要对所有令牌进行两层嵌套循环，并且必须确保该条目不对其自身计数。内置的 `enumerate` 函数允许我们在两层循环中跟踪条目的索引，跳过相同的索引而不是条目（如果条目在文本中重复出现，则该条目实际上可能共现）。`collections` 库中的 `Counter` 是一个有用的数据结构，它本质上是字典，默认值是 `int`。然后，reducer 需要对字典中的每个元素进行求和，计算 mapper 中所有计数器的总数。虽然输入相同，但现在的输出更紧凑：

```
run, ((run, 6), (see, 3), (spot, 6))
see, ((run, 3), (spot, 2))
spot, ((run, 6), (see, 2), (spot, 1))
```

stripe 方法不仅在其表示上更紧凑，而且也生成更少、更简单的中间键，从而优化了数据的 sort、shuffle 等方面。然而，stripe 对象更庞大，在处理时间和序列化方面的开销都更大，特别是当 stripe 非常大时。stripe 的大小有上限，尤其当共现矩阵非常密集时，可能需要大

量的内存来记录一个条目的数据。

介绍完 pair 和 stripe，键计算的讨论也就结束了。正如你看到的，大多数大数据计算都靠基于键的计算来提供和维护数据集之间的关系，以确保数据能合理分布在不同的 mapper 和 reducer 上。在 Spark 和 MapReduce 上执行大规模计算需要我们换个角度思考标准计算的传统方法，因为数据的规模实在太大。下一节将基于这种思维转变，提出具体的设计模式，这些模式通常用于分解以实现最后一英里计算。

5.2 设计模式

设计模式是软件设计中的一个特殊术语，是指针对特定编程挑战的通用、可重用解决方案。设计模式通常不限定语言，不仅指模式的实现细节，更指设计或构造策略。最常见的软件设计模式可能是在 Web 开发中非常流行的模型 - 视图 - 控制器（model-view- controller，MVC）模式，可以使用 Ruby、Java 等各种语言实现。

与之类似，我们也可以探索函数式设计模式，用于解决 MapReduce 和 Spark 中的并行计算问题。这些模式展示了可用于更复杂或特定于领域的角色的通用策略和原则。事实上，我们已经在用于计算单词共现的 pair 和 stripe 模式中看过一个例子。pair 和 stripe 都可以应用于更一般的计算。

在《MapReduce 设计模式》中，Donald Miner 和 Adam Shook 探索了 MapReduce 作业的 23 种常见设计模式。他们将模式大致分为如下几类。

概要

　　根据聚合、分组、统计度量、索引或数据的其他高级视图，提供大型数据集的摘要视图。

过滤

　　基于一组固定的条件创建数据的子集或样本，并且不以任何方式修改原始数据。

数据组织

　　将记录重组为有意义的模式，不一定使用分组。此任务适合作为后续计算的第一步。

连接

　　从不同来源将相关数据收集到一个统一的整体。

元模式

　　为复杂或优化过的计算实现作业链和作业合并。这类模式是与其他模式相关联的模式。

输入和输出

　　使用数据处理模式对输入源的数据进行转换，以输出到一个不同的输出源。输入源和输出源可以位于 HDFS，也可以是其他数据源。

本节将详细探讨 MapReduce 和 Spark 的概要和过滤技术，并概述 MapReduce 数据流和作业链的构造。我们将用几个具体应用来揭示这些模式背后的秘密，但有一点非常显而易见：这些技术通常将输入数据分解或转换成用于最后一英里计算的较小数据源。

5.2.1　概要

概要尝试用尽可能简单的方法描述尽可能多的关于数据集的信息。我们习惯于阅读内容提要（executive summary），它突出冗长文件的主要内容，而不涉及细节。与之类似，描述性统计数据试图通过测量观测数据的集中趋势（平均值、中值）、分散情况（标准差）、分布形状（偏度）或变量之间的依赖关系（相关性）来概括观察数据之间的关系。

基于键的计算对数据进行分组（另一种形式的概要），聚合某个通常能描述键的值（这可能有价值），然后概要计算就进入了下一步。基于键的计算可以是简单的分析，例如计算最不准点的机场或航空公司；也可以是稍复杂些的分析，例如推断天气、距离或其他因素对机场或航空公司的影响。在很多情况下，概要只是更大的概括和预测的第一步，例如作为语言模型的单词共现的计算，或描述概率分布的频率分析。

从原理上讲，MapReduce 和 Spark 是应用一系列的概要，将具体的数据形式（每个单独的记录）转换为更一般的形式。大体来说，我们最熟悉的概要具备以下操作特征：

- 聚合（集合到单个值，如平均值、总和或最大值）

- 索引（将值映射到值集合的函数映射）
- 分组（将集合选择或划分成多个集合）

在本节中，我们将探索聚合和索引的模式（通过之前讨论的基于键的技术可以轻松实现分组）。具体来说，我们将探索数据集的并行统计描述，例如 Pandas 中的 `describe` 命令以及如何实现聚合；然后，将探索两种索引技术：倒排索引和通过词频 - 逆文档频率（term frequency-inverse document frequency，TF-IDF）的文档概要。

01. 聚合

MapReduce 和 Spark 上下文中的聚合函数拥有两个输入值并生成一个输出值，它也满足交换律和结合律，因此可以并行计算。回顾一下，交换律表明顺序对二元运算没有影响，例如对给定的运算 ✣ 来说，`a ✣ b = b ✣ a`；结合律表示无论输入如何分组，计算都是相同的，即（`a ✣ b`）✣ `c = a ✣`（`b ✣ c`）。加法和乘法是满足交换律和结合律的，但减法和除法不是。

聚合一般是对一个集合应用操作以创建较小集合（聚集在一起），而 reduce 通常被认为是将集合 reduce 为单个值的操作。聚合也可以被认为是一系列更小的 reduce 组成的应用程序。在这种情况下，为什么结合性和交换性是实现并行不可或缺的条件也就显而易见了：给定一个 reduce `a ✣ b ✣ c ✣ d`，由于网络或其他物理约束导致 shuffle 的结果不确定，这意味着顺序不能有影响。结合性允许一个进程计算 `a ✣ b`，另一个进程计算 `c ✣ d`，其中一个进程发送它们的结果以执行最终的 ✣ 操作。

想想标准数据集描述量：平均值、中值、众值、最小值、最大值、总和。其中，总和、最小值和最大值都容易实现，因为它们都满足结合律和交换律；但平均值、中值和众值不是。对中值和众值来说，通常需要先进行某种排序，而由于涉及除法，分组计算平均值会产生精度损失。虽然这些计算有并行的近似算法，但更重要的是意识到，在执行这些类型的分析时应格外小心。我们将使用一个 MapReduce 作业来描述整个数据集，而不是单独讨论这些计算。

02. 统计概要

现在，可以通过两个关键概念来分析处理数据集。首先，使用

键作为关系来定义有意义的数据子集；其次，使用多种方法，包括作业链、键空间管理或 pair、stripe 等机制，同时实现多个计算。通过将实例按键分组并描述属性（类似于 Pandas 中的 `describe` 命令），可以简化大型数据集并对其执行概要分析。

我们已经见过了按键计算均值和计数的例子，这些类型的分析通常最先运行，从而对较大数据集有初步了解。特别是对高速变化的大数据（以惊人速度变化的数据）来说，运行定期描述性作业可以让你了解已经发生的变化以及它们是如何变化的。我们将在一个批次中一次性运行所有的 6 个作业，包括计算总数、总和、平均值、标准差和范围（最小值和最大值），而不是为每个描述性指标单独实现一个 MapReduce 作业（代价高昂）。

这时的基本策略是，为所有按键进行的计算映射一个计数器值的集合。然后，reducer 将每个操作分别应用于值集合中的每个项目，使用每个项目来计算最终输出（例如平均值取决于总数和总和）。这样的 mapper 的基本结构如下所示：

```
class StatsMapper(Mapper):

    def map(self):
        for key, value in self:
            try:
                value = float(value)
                self.emit(key, (1, value, value ** 2))
            except ValueError:
                    # 无法解析，忽略
                    pass
```

在这种情况下，直接进行 reduce 的三种操作是计算总数、总和和平方和。因此，该 mapper 针对每个键发射一次，1 用于计数，值用于求和，值的平方用于平方和。reducer 使用总数及总和来计算平均值，使用值计算范围，使用总数、总和以及平方和计算标准差，如下所示：

```
from ast import literal_eval as make_tuple

class StatsReducer(Reducer):

    def reduce(self):
        for key, values in self:
            # 解析mapper发送过来的值
```

```
        values = make_tuple(values)

        count   = 0
        delay   = 0.0
        square  = 0.0
        minimum = None
        maximum = None

        for value in values:
            count  += value[0]
            delay  += value[1]
            square += value[2]
            if minimum is None or value[1] < minimum:
                minimum = value[1]

            if maximum is None or value[1] > maximum:
                maximum = value[1]

        mean   = delay / float(count)
        stddev = math.sqrt((square-(delay**2)/count)/count-1)

        self.emit(key, (count, mean, stddev, minimum, maximum))
```

这个作业举例说明了如何将复杂数据类型作为 MapReduce 的输出和中间值使用，这是迈向高级分析方法的第一步。reducer 使用 `ast.literal_eval` 反序列化机制来解析值元组，然后对数据值执行单个循环（你必须将所有值加载到内存中，例如作为列表，以进行多次遍历）来计算各种和、最小值和最大值。

MapReduce 中的 reducer 可以访问与单个键相关联的所有值的可迭代对象，而在 Spark 中，必须对这个计算稍作修改。Spark 中的 reduce 不能应用以集合作为输入的操作，因而你必须每一次将操作应用于输入中的一对元素。又因为第一次应用的结果是第二次应用的第一个输入，所以操作必须满足结合律和交换律。例如，对于给定输入 `[5, 2, 7]`，你不能简单地将 `sum` 应用于集合，而要这样使用 `add`：`add(add(5, 2), 7)`。因此，必须为 mapper 输出的值添加最小值和最大值计数器，以分别记录 reduce 过程中的最小值和最大值，如下所示：

```
def counters(item):
    """
    将键值对解析为键和概要计数器
    计数器格式： (count, total, square, minimum, maximum)
    """
    key, value = item # 分解item元组
```

```
    try:
        value = float(value)
        self.emit(key, (1, value, value ** 2, value, value))
    except ValueError:
        # 无法解析，忽略
        pass

def aggregation(first, second):
    """
    对两个(key, counter)执行概要聚合
    """
    count1, total1, squares1, min1, max1 = first
    count2, total2, squares2, min2, max2 = second
    minimum = min((min1, min2))
    maximum = max((max1, max2))
    count   = count1 + count2
    total   = total1 + total2
    squares = squares1 + squares2

    return (count, total, squares, minimum, maximum)

def summary(aggregate):
    """
    根据聚合结果，计算概要统计
    """
    (key, (count, total, square, minimum, maximum)) = aggregate

    mean   = total / float(count)
    stddev = math.sqrt((square-(total**2)/count)/count-1)

    return (key, (count, mean, stddev, minimum, maximum))


def main(sc):
    """
    Spark应用程序的主要分析过程
    """

    # 给定一个键值对数据集，映射到counters
    dataset = dataset.map(counters)

    # 根据键执行概要聚合
    dataset = dataset.reduceByKey(aggregation)
    dataset = dataset.map(summary)

    # 将结果写入磁盘
    dataset.saveAsTextFile("dataset-summary")
```

`reduceByKey` 函数的规则使得 Spark 作业中的数据流不太一样。我们无法通过迭代跟踪最小值和最大值，而只能在计算结果中标注出最后看到的最小值和最大值，并在继续 reduce 时传

播它们。因此，我们不能在聚合期间简单地执行最终计算，而是需要另一个 map 在聚合后的 RDD（小得多）上完成概要计算。

这个 `describe` 示例提供了一种有用的模式，可以同时计算多个特征并将它们作为向量返回。这种模式经常被重用，在机器学习上下文中尤其如此，因为可能需要多个过程生成训练所需的实例（例如二次计算、归一化、插补、连接或更多具体的机器学习任务）。理解 MapReduce 聚合和 Spark 聚合之间的差异，对跟踪错误以及在 MapReduce 和 Spark 之间进行代码移植大有帮助。

5.2.2 索引

与基于聚合的概要技术不同，索引采用多对多的方法。聚合将多个记录收集到单个记录中，而索引将多个记录与一个或多个索引相关联。在数据库中，索引是用于快速查找的专用数据结构，通常是二叉树（binary-tree，B-Tree）。在 Hadoop/Spark 中，索引也能发挥类似的功能，但是它们不会被维护和更新，而通常会成为需要快速查找的下游计算的第一步。

文本索引在 Hadoop 算法“万神殿”中地位特殊，这是由于 Hadoop 最初被用于创建搜索应用程序。当仅处理一小部分文档时，它可以像 `grep` 一样扫描文档来查找搜索项。然而，随着文档和查询的数量增加，再使用这种方法就不合适了。在本节中，我们将看到两种类型的基于文本的索引：常见的倒排索引以及词频 - 逆文档频率（TF-IDF）。TF-IDF 是与索引相关联的数字统计量，通常用于机器学习。

01. 倒排索引

倒排索引是从索引项到文档集合中的位置的映射（与从文档到索引项映射的前向索引相反）。在全文搜索中，索引项是搜索项，通常是去除了停用词（例如在搜索中无意义的常见词）的词或数字。大多数搜索引擎还采用了某种词干提取（stemming）和词形还原（lemmatization）：具有相同含义的多个词被分类到单个词类（例如“running”“ran”“runs”由单个词语“run”索引）。

最常见的倒排索引用例是搜索：它让搜索算法能快速检索出要排列和返回的文档子集，而免于扫描每个文档。例如，要想查询“running bear”，可以用索引查找包含搜索项“running”和包

含搜索项“bear”的文档的交集。然后采用简单的排名系统来返回搜索项紧挨着，而不是相距很远的文档（现代搜索排名系统显然比这要复杂得多）。

然而，搜索示例可以被一般化到机器学习上下文。索引项不一定是文本，它可以是一条更长的记录的任意部分。此外，使用索引来简化或加快下游计算（如排名）的任务也很常见。根据索引的创建方式，可以在性能和准确率之间进行权衡；或者在给定随机索引的情况下，在精确率和召回率之间进行权衡。

来考虑某个预处理后的文本，其中文档 ID 和行号作为键，行文本作为值。这种预处理方式可以用于所有用户驱动的文本，如留言板或评论；但在这里，我们将它用于莎士比亚戏剧全集。具体来说，我们要创建一个人物关联索引。因此，不能将人物映射到已有的行，而是要针对人物和起始行进行概要分析，以便看出人物的出场顺序。语料库中的每一行表示如下：

```
hamlet@15261    HAMLET    O, that this too too solid flesh would mel
hamlet@15261              Thaw and resolve itself into a dew!
```

行的第一部分是 `title@lineno`（戏剧名 @ 行号）标识符，然后是一个 TAB 字符（`\t`）、人物的名字、第二个 TAB 字符和剧本中的一行文本。如果相同的人物连续说了多行，则用两个 TAB 字符将标识符与文本分开。为了创建一个人物的倒排索引，我们将使用一个恒等 reducer 和下面的 mapper（注意 Spark 中很容易实现相同的算法）：

```
class CharacterIndexMapper(Mapper):

    def map(self):
        for row in self:
            row = row.split("\t")              # 使用制表符拆分
            if not len(row) >= 3: continue     # 确保数据格式

            if row[1] != "":
                # 如果存在人物，发射名字和docid/lineno
                self.emit(row[1], row[0])
```

这个莎士比亚人物索引示例说明了索引的几个关键点。首先，索引项可以是任意的（这里是人物名称）；其次，这种算法虽然非常简

单，但它高度依赖于输入数据的结构（例如，我们知道要搜索 TAB 分片以找到人物名称）；最后，这个示例还使用了我们之前看到的一些 map 和 reduce 模式——恒等 reducer。继续发展的话，这个数据结构可以创建人物之间的对话图，更可以用于分析人物所在的人际圈或人物相似性。最关键的一点是，倒排索引通常是下游计算的第一步。

人物索引作业的输出是人物名称的列表，每个人名对应该人物每次开始说话的行的列表，这可以当作查找表或作为其他类型分析的输入使用。

02. **TF-IDF**

词频 - 逆文档频率（TF-IDF）可能是目前最常用的基于文本的概要形式，也是基于文本的机器学习中最常用的文档特征。TF-IDF 是一种指标，定义词条（单词）和作为较大语料库一部分的文档之间的关系。具体来说，它给出该词在其他文档中的相对频率，从而试着定义该词对于特定文档的重要性。

词频 $tf_{i,j}$ 是给定词条 i 在文档 j 中出现的次数，通常用于衡量该词与该文档的相关性。以一份关于美国政治的文件为例：一方面，我们可能会说像“民主”（democracy）或“选举”（election）这样的词语比“鲁米那”（luminal）这样的词语出现得更频繁，因此它们与文档的整体论述更相关；另一方面，词频本身将过度强调常见的词语，如“说”（speaking）——在给定组合语料库中，该词会出现于科学和政治类文档中。因此，词条 i 的文档频率 df_i，即该词条在多少文档中出现过，用于弥补词频的片面性。也就是说，包含词条的文档数与文档总数 N 的比例的倒数的对数与词频相乘。TF-IDF 分数高，则给定词经常在目标文档中出现，但不常在语料库的其他位置出现。文档 j 中的词条 i 的 TF-IDF 如下所示：

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

这种方法最初用于文档的主题建模，这是一种试图将主题相同的文档相互关联的聚类形式。不难看出，若文档共享高 TF-IDF 值的词，它们则可能彼此相关，因为这些词条不常出现在语料

库的其余部分。出于类似的原因，TF-IDF 现在被广泛应用于其他机器学习任务，包括分类、自动问答，甚至非结构化数据的社交网络或 Web 分析中。

在索引中加入该算法，原因和加入简单的倒排索引类似：它创建了通常用于下游计算和机器学习的数据结构。此外，这个更复杂的示例突出了其他几节只简单涉及的内容：使用作业链实现单个算法。考虑到这一点，让我们来看看 TF-IDF 的 MapReduce 实现。

我们的策略是使用键空间模式在三个作业中传播所需的数据：第一个作业使用简单的单词计数来计算每个文档的词频，该单词计数还维护该词的文档 ID；第二个作业计算该词一共出现在多少文档中；最后一个作业使用前两个作业传播到最后的信息计算 TF-IDF。第一个作业如下所示：

```
class TermFrequencyMapper(Mapper):

    def __init__(self, *args, **kwargs):
        """
        初始化分词器和停用词
        """
        super(TermFrequencyMapper, self).__init__(*args, **kwargs)

        self.stopwords = set()
        self.tokenizer = re.compile(r'\W+')

        # 从文本文件读取停用词
        with open('stopwords.txt') as stopwords:
            for line in stopwords:
                self.stopwords.add(line.strip())

    def tokenize(self, text):
        """
        对一行文本进行分词和规范化（只产生非数字、标点和空字符串的非停用词）
        """
        for word in re.split(self.tokenizer, text):
            if word and word not in self.stopwords and word.isalpha
                yield word

    def map(self):
        for docid, line in self:
            # 对每一行分词，并发射每个(word, docid)
            for word in self.tokenize(line):
                self.emit((word, docid), 1)

class SumReducer(Reducer):
```

```
    def reduce(self):
        for key, values in self:
            total = sum(int(count[1]) for count in values)
            self.emit(key, total)
```

为了计算文档中的词，不能简单地使用空格分割行，而是要使用正则表达式对文本进行分词——这可能会因为索引需要而变得更复杂。我们还从 stopwords.txt 文件中读取了停用词列表，该文件需要包含在作业中。因此，我们的分词方法简单地使用正则表达式进行拆分，并过滤掉停用词、数字和标点符号。更高级的分词器也可以提取词干，或者实现归一化（例如全部变为小写）。第一个作业发射 (term, docid) 为键、频率为值的元组。

第二个作业由一个 mapper 和一个 reducer 组成，如下所示：

```
class DocumentTermsMapper(Mapper):

    def map(self):
        for line in self:
            key, tf = line.split(self.sep) # 将每一行拆分成键值对
            word, docid = make_tuple(key) # 解析元组字符串
            self.emit(word, (docid, tf, 1)) # 发射词和带计数器的数据

class DocumentTermsReducer(Reducer):

    def reduce(self):
        for word, values in self:
            # 将values加载到内存，进行多次处理和解析
            values = [make_tuple(value) for value in values]

            # 第一次处理：计算词条的文档频率
            terms = sum(int(item[2]) for item in values)

            # 第二次处理：为与docid关联的每一个word发射一个值
            for docid, tf, num in values:
                self.emit((word, docid), (int(tf), terms))
```

此作业的 mapper 又是一个计数 mapper，用于求词条的文档频率的和；它还改变了键空间，维护针对该文档的词频并将文档 ID 添加到值中。这样，我们可以按词 reduce，其中每个值都对应一个文档。因此，reducer 需要遍历数据两次：一次求和，另一次执行每个文档的键空间更改。为了做到这一点，必须将元组 (docid, tf, count) 缓存在内存中，使用列表推导从生成器加载数据。如果许多文档都包含该词（如“the”这样的

高频词），这个计算也许不能在内存中进行。也正因如此，停用词列表对 TF-IDF 的计算才如此重要。其他解决方法包括：将中间数据临时存储到磁盘；再实现一个中间 MapReduce 作业，一个作业用于求词条的文档频率的和，另一个用于改变键空间：

```
class TFIDFMapper(Mapper):

    def __init__(self, *args, **kwargs):
        self.N = kwargs.pop("documents")
        super(TFIDFMapper, self).__init__(*args, **kwargs)

    def map(self):
        for line in self:
            key, val = map(make_tuple, line.split(self.sep))
            tf, n = (int(x) for x in val)
        if n > 0:
            idf = math.log(self.N/n)
            self.emit(key, idf*tf)
```

最后一个作业是一个只有 map 的作业，因为我们已经有了计算用的键——由上一个 reducer 发射的 `(word, docid)` 对。使用恒等 reducer 就完全能够搞定。我们简单地将行解析为 int 的元组，并且只要频率大于零，就计算 TF-IDF。请注意，需要一条额外的信息——语料库中的文档数量，它在这个过程中没有参与计算。

这个任务虽然看似很复杂，但是将执行过程设想为数据流就能好很多：随着计算结果片段的产生，它们被添加到数据流中。键／值选择由计算中的下一步骤激发。而且最重要的是，这个计算仅仅遍历了一次原始输入，所以它支持作业的线性依赖。TF-IDF 计算的 Spark 实现也需要这种数据流思维模式，如下所示：

```
def tokenize(document, stopwords=None):
    """
    分词并返回(docid, word)和一个计数
    """

    def line_tokenizer(lines):
        """
        逐行分词的内部生成器
        """
        for line in lines:
            for word in re.split(tokenizer, line):
```

```
                if word and word not in stopwords.value and word.isa
                    yield word

    docid, lines = document
    return [
        ((docid, word), 1) for word in line_tokenizer(lines)
    ]

def term_frequency(v1, v2):
    """
    拆分复杂的值，计算词频
    """
    docid, tf, count1 = v1
    _docid, _tf, count2 = v2
    return (docid, tf, count1 + count2)

def tfidf(args):
    """
    给定((word, docid), (tf, n))参数，计算TF-IDF
    请注意，必须提前定义N_DOCS，它是语料库中的文档数（n是word的文档频率）
    """
    (key, (tf, n)) = args
    if n > 0:
        idf = math.log(N_DOCS/n)
        return (key, idf*tf)

def main(sc):
    """
    Spark应用程序的主要分析过程
    """

    # 从数据集加载停用词
    with open('stopwords.txt', 'r') as words:
        stopwords = frozenset([
            word.strip() for word in words.read().split("\n")
        ])

    # 将停用词广播到集群
    stopwords = sc.broadcast(stopwords)

    # 第一阶段： 分词并计算文档频率
    # 请注意： 假设有一个包含(docid, text)对的语料库
    docfreq = corpus.flatMap(partial(tokenize, stopwords=stopwords))
    docfreq = docfreq.reduceByKey(add)

    # 第二阶段： 计算词频，然后执行键空间更改
    trmfreq = docfreq.map(lambda (key, tf): (key[1], (key[0], tf, 1)
    trmfreq = trmfreq.reduceByKey(term_frequency)
    trmfreq = trmfreq.map(
        lambda (word, (docid, tf, n)): ((word, docid), (tf, n))
    )

    # 第三阶段：为每个(word，document)对计算TF-IDF
```

```
    tfidfs = trmfreq.map(tfidf)
```

Spark 作业同样从磁盘加载停用词，然后将其广播到集群的剩余部分。然后，就可以对默认参数为停用词广播值的 `tokenize` 偏函数应用 `flatMap` 操作。这里之所以使用 `flatMap`，是因为 `tokenize` 函数将为文档中的每一行生成一个令牌计数列表（需要使用内部的 `line_tokenizer` 函数）。最后，将 Spark 实现的 `term_frequency` 和 `tfidif` 函数映射到每个文档。请注意，因为 `reduceByKey` 被调用了两次，并且需要在 `tfidfs` RDD 上应用某个最后的动作，所以此 Spark 作业同样具有三个数据流，和 MapReduce 作业一样。

5.2.3　过滤

过滤是粗粒度地减少下游计算数据的主要方法之一。与聚合通过宏观概览分组来缩小输入空间不同，过滤意在通过去除不需要的记录来缩小计算空间。在键空间一节中，我们探讨了应用于 mapper 的过滤。事实上，因为 mapper 非常适合执行过滤，所以许多过滤任务常使用只有 map 的作业（不需要 reducer）。这可以视为通过谓词或选择进行过滤，类似于 SQL 语句中的 `where` 子句。

另外一些过滤任务使用 reducer，收集具有代表性的数据集或根据值进行过滤。这种过滤包括查找最大的 `n` 个值或最小的 `n` 个值、去重或子选择。分析中非常常见的过滤任务是抽样：创建一个较小的、具有代表性的数据集，该数据集相对于较大的数据集分布良好（取决于你期望实现的分布类型）。开发中使用面向数据的子样本验证机器学习算法（例如交叉验证）或者进行其他统计计算（例如幂运算）。

我们通常可以将过滤实现为一个函数，它接受一条记录作为输入。如果评估返回 true，则发射记录，否则丢弃记录。本节将探讨无序的最大 / 最小 *n* 个元素、抽样技术，以及经布隆过滤器提高性能后的高级过滤。

01. **top n**记录

top `n` 记录（以及相反的 bottom `n` 记录）方法是一个基数比较过滤器，它需要一个 mapper 和一个 reducer。其基本原理是让每个 mapper 产生其 top `n` 个项目，然后 reducer 将从 mapper 产生的项目中同样选择 top `n` 个项目。如果 `n` 相对较

小（至少与数据集的其余部分相比），单个 reducer 应该能够轻松处理该计算，因为每个 mapper 最多产生 `n` 条记录：

```
import bisect

class TopNMapper(Mapper):

    def __init__(self, n, *args, **kwargs):
        self.n = n
        super(TopNMapper, self).__init__(*args, **kwargs)

    def map(self):
        items = []
        for value in self:
            # 维护有序的items列表
            bisect.insort(items, value)

        for item in items[-self.n:]:
            # 从mapper发射前n个值
            self.emit(None, item)

class TopNReducer(object):

    def __init__(self, n, *args, **kwargs):
        self.n = n
        super(TopNReducer, self).__init__(*args, **kwargs)

    def reduce(self):
        items = []
        for _, values in self:
            for value in values:
                bisect.insort(items, value)

    for item in items[-self.n:]:
        # 从mapper发射前n个值
        self.emit(None, item)
```

这里的 mapper 和 reducer 都使用了 `bisect` 模块将值按升序插入列表。为了获得最大的 *n* 个值，使用了负索引的切片，从而选择有序列表中的最后 *n* 个值。要得到最小的 *n* 个值，可以简单地分片取出列表中的前 *n* 个值。使用 `None` 作为键可确保仅使用单个 reducer。请注意，Spark 拥有丰富的 RDD API，你可以使用 `top` 和 `takeOrdered` 动作，而不必自己实现。请注意，对于 Spark 和 MapReduce，为了能进行排序，需要记录是可比较的，这需要进行严格的解析；例如，在 Python 中，`'14'> 22` 为 `True`。

这种方法最大的好处是不需要对整个数据集进行完整的排序；相反，每个 mapper 对它们自己的数据子集进行排序，而 reducer 仅看到 mapper 数量 `n` 倍的数据。这段代码可以通过几种方式进行优化，但主要优化与所使用的数据结构有关。下一节将研究如何在 `bisect` 模块上使用堆（`heap`）来实现类似的功能。

02. 简单随机抽样

简单随机抽样是数据集的子集，数据集的每条记录属于该子集的可能性相同。在这种情况下，评估函数不关心记录的内容或结构，而是利用某种随机数生成器来评估是否发射记录。但问题来了，如何确保每个元素被选中的可能性相同呢？

如果需要的样本不是必须大小为 `n`，而是包含百分之多少的记录就好，第一种方法是简单地使用随机数生成器来产生数字，并将其与期望的阈值大小进行比较。随机数生成器可用值的范围与阈值将共同决定大约发射百分之多少的记录。一般来说，随机数生成器返回的值在 0~1，因此与百分比的直接比较将产生预期的结果！例如，如果想要从数据集中采样 20%，可以写如下的 mapper：

```
import random

class PercentSampleMapper(Mapper):

    def __init__(self, *args, **kwargs):
        self.percentage = kwargs.pop("percentage")
        super(PercentSampleMapper, self).__init__(*args, **kwargs)

    def map(self):
        for _, val in self:
            if random.random() < self.percentage:
                self.emit(None, val)

if __name__ == '__main__':
    mapper = PercentSampleMapper(sys.stdin, percentage=0.20)
    mapper.map()
```

使用 `percentage` 关键字参数初始化 `PercentSampleMapper`，该参数从 `__init__` 中的通用关键字参数中取出。此作业将返回原始数据集的 20%左右，因为每条记录产生的随机数小于 0.2 的可能性相同，所以随机数小于 0.2 发生的可能性只大约为调用次数的 20%。如果这个作业

运行时只有 mapper 而没有 reducer，许多小文件将被写入到磁盘，文件的数量与 mapper 的数量相同。使用一个恒等 reducer 将确保这些值都被收集到单个文件中。

然而，如果想要一个大小精确为 n 的样本呢？为了确保每种方法机会均等，必须进行 n 次随机选择，每次选择一个元素，而不进行替换，以确保每条记录被选中的机会均等。一种方法是打乱记录，选择 0～N-1 的一个随机数，其中 N 是记录数，并发射在该索引处的记录。然后再次打乱，选择 0～N-2 的随机数，依此类推。

> 统计学家可能倾向于采用蓄水库抽样（reservoir sampling）技术，它能够在单进程上下文中对数据流或大型数据集进行高效采样。一般来说，当你在 mapper 中使用任何概率分布时，你都必须小心，因为不能保证 mapper 在多次运行中看到相同的数据，也不能保证每个 mapper 获得相同数量的数据，更不能保证映射过程保持一个特定的顺序。这些不确定性会导致一些 mapper 预期的可能性过高或过低。对此（正确）的反应应该是将工作移动到一个 reducer 或聚合上进行，但这样做的话，在集群上多进程执行的优势可能会不复存在！虽然有分布式的蓄水库抽样算法，但是要谨记，同一算法的串行和并行实现往往可能迥然不同！

为了并行化打乱（shuffle）方法，我们可以想象将一幅扑克牌平均发给了四位玩家。如果想要抽取 4 张牌，并且让每张牌被选中的可能性相等，可以简单地让每位玩家洗他们各自手中的牌，然后每个人给你发 4 张牌；然后，你从这 16 张牌中选择前 4 张。但如果你把牌从空中掷给每位玩家，而不是平均发给他们，则每位玩家得到的牌数可能不均等，但这种方法仍然能确保每张牌被选中的可能性相等。这时问题就变成了：该如何使用 Hadoop 来打乱记录，以便获得更好的性能？

答案是在 mapper 中为每个记录分配一个 0～1 的随机浮点数。随后，mapper 会发射前 n 个记录。同样，reducer 也只会发射它从 mapper 接收的前 n 个记录。虽然此机制仍然只允许单个 reducer，但是该 reducer 获取的是数据的有限子集（例如 mapper 数量的 n 倍），子集应该能够放入 reducer 的内存

中。因为每行拥有 n 个最大随机数之一的概率相等，所以能获得一个随机样本：

```
import random, heapq

class SampleMapper(Mapper):

    def __init__(self, n, *args, **kwargs):
        self.n = n
        super(SampleMapper, self).__init__(*args, **kwargs)

    def map(self):
        # 将堆初始化成一个包含n个0的列表
        heap = [0 for x in xrange(self.n)]

        for value in self:
            # 维护一个堆，只包含最大的n个值
            heapq.heappushpop(heap, (random.random(), value))
            for item in heap:
                # 发射抽样数据
                self.emit(None, item)


class SampleReducer(Mapper):

    def __init__(self, n, *args, **kwargs):
        self.n = n
        super(SampleReducer, self).__init__(*args, **kwargs)

    def reduce(self):
        # 将堆初始化为一个包含n个0的列表
        heap = [0 for x in xrange(self.n)]

        for _, values in self:
            for value in values:
                heapq.heappushpop(heap, make_tuple(value))

        for item in heap:
            # 发射抽样数据
            self.emit(None, item[1])
```

我们本可以像使用 top n 记录方法时一样使用 bisect 模块，但是为了获取多样性，我们使用堆数据结构在内存中维持一个只有 n 个最大随机值的列表。这进一步减小了 mapper 和 reducer 的内存需求（对 reducer 尤为明显），使每次只有 n 个值保存在内存中。我们的 mapper（reducer 也类似）初始化了一个长度 n 值为零的列表。heapq.heappushpop 函数将新值压入到堆中，然后弹出最小值（而且还比顺序调用

heapq.push 和 heapq.pop 快得多）。

03. 布隆过滤

布隆过滤器是一种高效的概率型数据结构，用于执行集合成员资格测试。布隆过滤器与其他评估函数没有什么不同，除了一点：它必须进行预先计算以收集“热值”（排除集的成员）——需要过滤的值。布隆过滤器的好处在于，它很紧凑（方便将大集合传输到集群上的每个 mapper），并且能快速测试成员资格。

但是，布隆过滤器可能会误判（false positive）；换句话说，它会把不属于集合的元素判断为属于集合。不过，它能保证不会排除任何属于集合的元素——没有漏报（false negative）。因此，表达式 x in bloom 就意味着“x 可能在集合中”或“x 绝对不在集合中”。这也决定了布隆过滤器的构造，因为你要在过滤器集合的大小、数据中可能有多少元素，以及 mapper 和 reducer 的内存容量之间进行权衡。通过权衡，你能采用不同程度的模糊性。在构造大多数布隆过滤器时，可以设置误判的概率阈值，这会使布隆过滤器增大或缩小。

使用布隆过滤器的第一步是构建它。布隆过滤器对输入数据应用几个散列函数，然后根据散列值设置位数组中的位。一旦构建了位数组，就可以将散列函数应用于测试数据并查看相关位是否为 1，从而测试成员资格。根据一定的规则将不同的值映射到构造布隆过滤器的 reducer，可以并行化位数组的构造过程；位数组也可以是由其他进程维护的活动的版本化数据结构。

在这个例子中，我们将使用第三方库 pybloomfiltermmap，可以通过 pip 安装。虽然 Python 有很多第三方布隆过滤器库，但这个库提供了创建可配置过滤器的最好的 API。来思考一个例子：基于推文是否包含词条和用户名白名单中的标签（#）或者 @ 回复，决定是否包含推文。为了创建布隆过滤器，从磁盘加载数据，并将布隆过滤器保存到一个 mmap 文件，如下所示：

```
from pybloomfilter import BloomFilter

bloom = BloomFilter(1000000, 0.1, 'twitter.bloom')
for prefix, path in (('#', 'hashtags.txt'), ('@', 'handles.txt')):
    with open(path, 'r') as f:
```

```
        for word in f:
            bloom.add(prefix + word.strip())
```

本示例创建了一个有 100 万元素、错误率为 0.1 的布隆过滤器。它在底层使用这些参数来选择最优数 k（所需散列函数的数量），以保证给定容量下的错误阈值。性能和空间也存在折中——容量越小且错误率越低，需要的散列函数就越多，计算也就越慢；容量越大，布隆过滤器就必须越大。从磁盘文件读取标签和 Twitter 句柄（并给它们加上适当的前缀）后，布隆过滤器将被写入磁盘上一个名为 twitter.bloom 的文件中。

在 Spark 中使用它：

```
ELEMS = re.compile(r'[#@][\w\d]+')

def tweet_filter(tweet, bloom=None):
    for elem in ELEMS.findall(tweet['text']):
        if elem in bloom.value:
            return True

# 从磁盘加载布隆过滤器，进行并行化
bloom = sc.broadcast(BloomFilter.open('twitter.bloom'))

# 从磁盘加载JSON推文，进行解析
tweets = sc.textFile('tweets').map(json.loads)
tweets = tweets.filter(partial(tweet_filter, bloom=bloom))
```

我们的推文过滤器是使用 `functools.partial` 函数创建的，该函数创建了一个拥有布隆过滤器广播变量的闭包，而布隆过滤器是从驱动程序所处主机的磁盘加载的。`tweet_filter` 函数使用一个正则表达式提取所有标签和 @ 回复，然后检查它们是否在布隆过滤器中；如果是，则返回 `True`，从而保留 RDD 中与白名单匹配的所有元素。

布隆过滤器可能是常用于 Hadoop 分析的最复杂的数据结构。此处提到它不是因为它的复杂性，而是为了表明性能和正确性的结合将如何影响分布式计算。作为实践大数据的数据科学家，你会发现随机方法能助力及时计算，这是进一步分析所需要的。

5.3 迈向最后一英里分析

在本章中，我们研究了许多数据分析模式，从键计算到聚合、过滤的常规模式。这里面有一个宏观主题：将数据从较大的输入分解为较小的、更易于管理的输入。使用我们在本章中讨论的工具，本节将讨论端到端预测模型的计算策略。

许多机器学习技术在底层使用广义线性模型（generalized linear model，GLM）来估计给定输入数据和误差分布的响应值（response variable）。最常用的 GLM 是线性回归（还有逻辑回归和泊松回归），为模拟因变量 Y 和一个或多个自变量 X 之间的连续关系建模。该关系由一组系数和一个误差项表示如下：

$$Y = \beta_0 + \beta_1 X_1 + \cdots + \beta_n X_n + \epsilon$$

虽然只轻描淡写地说了说这个非常重要的话题，但是必须强调的是，系数 β 的计算是将模型拟合到现有数据的主要目标。这通常通过一个优化算法来完成——根据给定的某个数据集的 X 和 Y 的观察值，该算法能找到一组最小化错误量的系数。请注意，线性回归可以被认为是一种有监督机器学习方法，因为“正确”答案（拟合模型的 X 和 Y 变量）是预先已知的。

普通最小二乘法和随机梯度下降这样的优化算法是迭代的；也就是说，它们要多次遍历数据。在大数据环境中，每次优化迭代都多次读取完整数据集可能会非常耗时，在按需分析或开发中尤其明显。Spark 在 MLlib 中提供的分布式机器学习算法和内存计算让情况稍有好转，具体内容将在第 9 章讨论。但如果碰上极大的数据集或极小的时间窗口，即使是 Spark 也需要花费很长时间；如果 Spark 没有你想要实现的模型或分布式算法，那么分析方法的选择范围将因分布式编程的诸多困难而受限。

通用的解决方案是贯穿整章的内容：将输入数据集转换为更小的数据集，让它可以在内存中被处理，从而达到分解问题的目的。一旦数据集被缩小成内存计算，它就可以使用标准技术进行分析，然后在整个数据集中验证。对于线性回归，我们可以对数据集进行简单随机抽样，对样本执行特征提取，构建线性模型，然后通过计算整个数据集的均方误差来验证模型。

5.3.1　模型拟合

考虑一个具体的例子：我们有一个新闻报道或博文的数据集，现在要预测接下来的 24 小时内的评论数量。针对网络爬取的原始 HTML 页面，数据流应如下所示。

(1) 解析 HTML 页面获取元数据，并将主文本与评论分开。

(2) 创建一个从时间戳到博文评论 / 评论者的索引。

(3) 使用该索引为模型创建实例，实例是一篇博文以及 24 小时滑动窗口内的评论。

(4) 将实例与主文本数据（评论和博文）连接起来。

(5) 提取每个实例的特征（例如，前 24 小时的评论数、博文长度、从窗口到发布时间之间的时间、bag of words 特征、星期几，等等）。

(6) 对实例特征取样。

(7) 使用 Scikit-Learn 或 Statsmodels 在内存中构建一个线性模型。

(8) 计算整个数据集的实例特征的均方误差或者决定系数。

该数据流表明，许多预处理作业仅需要运行一次或几次（例如，特征提取需要在整个特征分析生命周期中反复运行）。然而，模型抽样和验证过程可以例行运行。一旦启动并运行这个模型，它甚至可以在线运行，当新的信息被馈送到数据流水线中时，该模型重新进行拟合和验证。

此时，假设通过所学技术，我们已经成功得到一个具有所有特征的数据集。使用本章前面介绍过的抽样技术，可以获取更小的数据集，将其保存到磁盘，并使用 Scikit-Learn 构建一个线性模型：

```
import pickle
import numpy as np

from sklearn import linear_model

# 从磁盘的制表符分割文件加载数据
data = np.loadtxt('sample.txt')

# 目标是第一列（键），X是值
y = data[:,0]
X = data[:,1:]

# 实例化并拟合线性模型
clf = linear_model.Ridge(alpha=1.0, fit_intercept=True)
clf.fit(X, y)

# 将模型作为pickle写入磁盘
with open('clf.pickle', 'wb') as f:
```

```
    pickle.dump(clf, f)
```

这段代码使用 np.loadtxt 函数从磁盘加载样本数据，在这个例子中是包含实例的制表符分隔文件，第一列是目标值，其余列是特征。这种类型的输出与 Spark 和 MapReduce 将键值对写入磁盘时的格式吻合，但是你必须将集群中的数据收集到单个文件，并确保文件格式正确。然后，数据被拟合到岭回归，这是一种使用正则化来防止过拟合的线性回归模型。

5.3.2 模型验证

为了在集群中评估这个模型的效果，我们有两个选择。第一种，将 Scikit-Learn 线性模型属性 clf.coef_（系数）和 clf.intercept_（错误项）写入磁盘，然后将这些参数加载到我们的 MapReduce 或 Spark 作业中，并自行计算误差。然而，这需要为每个模型实现一个预测函数。第二种，使用 pickle 模块将模型转储到磁盘，然后将其加载到集群中的每个节点供预测使用。现在就来编写 Scikit-Learn 模型误差估计模板，因为我们在进行假设驱动开发（例如调整参数、执行特征分析或模型选择），所以可以使用任意 Scikit-Learn 模型。

要想验证模型，就必须计算整个数据集的均方误差（mean square error，MSE）。误差被定义为实际值和预测值之间的差值 $y - \hat{y}$。为了确保没有负值（这将减少误差），我们将计算平方误差的均值。为此，只需要一个计算平均值的 reducer 和一个加载模型并计算均方误差的 mapper 即可：

```
import pickle

class MSEMapper(Mapper):

    def __init__(self, model, *args, **kwargs):
        super(MSEMapper, self).__init__(*args, **kwargs)

        # 从磁盘加载模型
        with open(model, 'rb') as f:
            self.clf = pickle.load(f)


    def map(self):
        for row in self:
            # 解析行内浮点数值
            row = map(float, row)
            y = row[0]
```

```
            X = row[1:]

            yhat = self.clf.predict(x)

            self.emit(_, (y-yhat) ** 2)
```

可以在 Spark 中使用一个累加器来求平方误差之和，并将模型广播到集群上，如下所示：

```
def cost(row, clf=None):
    """
    计算给定行的平方误差
    """
    return (row[0] - clf.predict(row[1:])) ** 2


def main(sc):
    """
    Spark应用程序的主要分析过程
    """

    # 从pickle文件加载模型
    with open('clf.pickle', 'rb') as f:
        clf = sc.broadcast(model.load(f))


    # 创建累加器，求平方误差的和
    sum_square_error = sc.accumulator(0)

    # 加载和解析博客数据
    blogs = sc.textFile("blogData").map(float)

    # 映射cost函数，累加平方误差
    error = blogs.map(partial(cost, clf=clf))
    error.foreach(lambda cost: sum_square_error.add(cost))

    # 计算平均平方误差并打印
    print sum_square_error.value / error.count()
```

使用 `pickle` 模块来序列化 Scikit-Learn 模型是使用极大数据集进行机器学习的好开始。工作流通常是将经过序列化的模型存储在数据库 blob 字段中，然后根据需要在集群中进行加载和验证。更高级的大数据和扩展需要像 Mahout 和 Spark 的 MLlib 这样的机器学习库，这些内容将在第 9 章进行详细讨论。当然，对于近期开发的没有对应的分布式实现版本的模型，或者不能并行化的模型来说，我们还是有办法的。无论采用哪种方式，抽样、训练、验证策略都是行之有效的分析方法。

5.4　小结

本书以描述大数据上下文中的数据科学流程开始，重点介绍了构建数据产品和数据科学流水线。然后就顺理成章地从这些一般性话题转向了更具体的分布式计算、MapReduce 和 Spark。在本章开头，你应该已经轻松理解了分布式计算的工作原理、如何在 Hadoop 集群上实现作业，但不一定了解要实现什么。本章旨在让你了解各种分布式计算模式，并介绍了一些分析方法，以说明如何将其他数据处理工作流改编成大规模分析的版本。

我们确定的第一件事就是使用键进行计算，这是一种自然而然的并行化技巧，使我们能同时在多个集或域上进行操作。基于键的计算允许将操作同时应用到多个集合，而不需要进行若干独立的查询。理解如何使用键计算对于理解 MapReduce 非常重要，与 Spark 计算也联系紧密。为此，给 mapper 和 reducer 引入几个基于键的模式，包含 MapReduce 和 Spark 两种实现。然后，本章继续讨论更高级的算法和操作的设计模式；研究了概要、索引和过滤模式，但在这个过程中提出了非常常见的分析，如 TF-IDF、描述和随机抽样。这些模式和算法呈现了怎样用 Hadoop 进行分析，而不是怎样使用 Hadoop 进行计算。最后，展示了一个使用“最后一英里计算”计算线性回归的端到端分析的案例。我们描述了分解输入域、执行内存计算、在集群中验证计算的基本初始策略，这种策略可应用于数据流水线的许多部分，在敏捷、假设驱动的开发工作流中支持各种分析。

本章是本书第一部分和第二部分之间承上启下的部分。上半部分讨论了 Hadoop 裸机（bare metal）和计算细节，下半部分更多地将 Hadoop 看作是一个数据管理工具。接下来的章节将重点介绍 Hadoop 生态系统和支持集群中数据流水线的工具。我们将研究使用 Hive 的数据挖掘和数据仓储、使用 Sqoop 的数据采集、使用 Pig 和 Spark 等高阶工具的数据流，以及使用 Spark MLlib 的机器学习。这一章是桥梁，连通从使用 Hadoop 裸机能实现的功能，到使用这些库可能实现的功能之间的路。

第二部分　大数据科学的工作流和工具

本书第二部分将探索更高级的工作流和工具，供数据科学家使用。虽然掌握了 Hadoop、MapReduce 和 Spark 的基础知识才能了解可以进行什么样的大规模分析，但是与大数据打交道的数据科学家通常每天都围绕着构建于 Hadoop 之上的工具生态系统工作。总体来说，第二部分围绕第 1 章提出的数据产品流水线组织章节内容。

第 6 章讨论数据挖掘和数据仓储，并就关系型和列式数据存储及查询分别介绍 Hive 和 HBase；第 7 章阐明对能将数据采集到 HDFS 的采集工具的需求，研究如何使用 Sqoop 采集结构化数据，以及如何使用 Flume 采集非结构化数据；第 8 章探讨用于分析的高级 API：Apache Pig 和 Spark DataFrame；第 9 章讨论使用 Spark MLlib 的机器学习和计算方法；最后，第 10 章对以上各章讨论过的工作流进行总结，并对数据科学进行全面回顾。

第 6 章　数据挖掘和数据仓储

作为数据分析师，我们通常更愿意专注于能获取有意义见解的数据挖掘任务，或者对经过整理（curated）、清洗和分段（staging）的数据应用预测建模方法。然而，在大多数传统企业的数据环境中，在进行任何有意义的数据分析之前，都需要投入大量的工程和技术资源来收集这些数据，并将其组织到统一的数据仓库中。

因此，企业数据仓库（enterprise data warehouse，EDW）已经成为大多数企业处理和分析大规模数据的关键。然而，由于绝大多数 EDW 使用某种形式的关系数据库管理系统（relational database management system，RDBMS）作为主要的存储工具和查询引擎，因此在开展新的数据分析项目时，将在前期模式设计和 ETL 操作上花费大量精力。据估计，ETL 将占数据仓储成本、风险和实施时间的 70%~80%。1 这种开销导致即使是最一般的数据分析原型设计或探索性分析也成本高昂。

> 1Kimball Group,“New Directions for ETL”(https://channels.theinnovationenterprise.com/articles/new-directions-for-etl).

当我们需要存储和分析的数据的数据类型急剧增加时——这些数据可以是非结构化的（电子邮件、多媒体文件）或半结构化的（点击流式数据）——RDBMS 就暴露了另一个局限性：数据的速度和多样性常

常需要“即时”地演进模式，这要被传统的 DW 支持是件非常困难的事。

正是出于这些原因，Hadoop 成为了数据仓储和数据挖掘领域最具革命性的技术。它将存储与处理分离，使公司能够将其原始数据存储在 HDFS 中，而不需要通过 ETL 将数据整合到一个统一的数据模型中。此外，通过使用 YARN 的通用处理层，我们能够从多个角度直接访问和查询原始数据，还能根据特定用例使用不同的方法（SQL、非 SQL）。因此，Hadoop 不仅支持探索性分析和数据挖掘原型设计，还为数据和分析的新类型打开了大门。

本章将介绍一些 Hadoop 中的主要框架和工具，用于实现数据仓储和数据挖掘功能；还将探索 Hadoop 最受欢迎的基于 SQL 的查询引擎 Hive，以及 NoSQL 数据库 HBase；最后，将再简单介绍一些数据仓储领域的著名 Hadoop 项目。

6.1　Hive结构化数据查询

Apache Hive 是一个建立在 Hadoop 之上的“数据仓储”框架。Hive 为数据分析人员提供了熟悉的、基于 SQL 的 Hadoop 接口，使他们能为 HDFS 中的数据添加结构化模式，并能使用 SQL 查询访问和分析该数据。Hive 使熟练使用 SQL 的开发人员能发挥 Hadoop 的可扩展性和弹性，而不需要他们学习 Java 或原生的 MapReduce API。

Hive 提供了自己的 SQL 方言，被称为 Hive 查询语言（Hive Query Language，HQL）。HQL 支持许多常用的 SQL 语句，包括数据定义语句（data definition statement，DDL，例如 `CREATE DATABASE/SCHEMA/TABLE`）、数据操作语句（data manipulation statement，DMS，例如 `INSERT`、`UPDATE` 和 `LOAD`）和数据检索查询（例如 `SELECT`）。Hive 还支持集成用户定义函数，这些函数可以由 Java 或 Hadoop Streaming 支持的任何语言编写，扩展了 HQL 的内置功能。

Hive 命令和 HQL 查询被编译成执行计划或一系列 HDFS 操作和／或 MapReduce 作业，然后在 Hadoop 集群上执行。因此，Hive 继承了 HDFS 和 MapReduce 的某些限制，无法提供传统数据库管理系统应有的关键联机事务处理（online transaction processing，OLTP）功能。具体来说，因为 HDFS 是写一次，读多次（WORM）文件系统，并且不提供就地文件更新，因此 Hive 执行起行级插入、更新或删除不是非常高效。事实上，这些行级更新最近才被 Hive 的 0.14.0 版本

（https://issues.apache.org/jira/browse/HIVE-5317）支持。

此外，为了满足在集群上生成和启动编译的 MapReduce 作业所需的开销，Hive 查询需要更长的延迟；在传统 RDBMS 上几秒就能完成的小型查询在 Hive 中可能需要几分钟才能完成。

好在 Hive 提供了所有基于 Hadoop 的应用程序都应有的高可扩展性和高吞吐量。因此，它非常适用于联机分析处理（online analytical processing，OLAP）的批处理任务，处理 TB 级甚至 PB 级的超大数据集。

本节将探讨 Hive 的一些主要功能，并编写 HQL 查询以执行数据分析。我们假设你已经在伪分布式模式的 Hadoop 上安装了 Hive，Hive 的安装步骤可参见附录 B。

6.1.1　Hive命令行接口（CLI）

Hive 的安装包里有一个方便的命令行接口（command-line interface，CLI），我们将使用它与 Hive 交互，并运行 HQL 语句。从 `$HIVE_HOME` 启动 Hive CLI：

```
~$ cd $HIVE_HOME
/srv/hive$ bin/hive
```

这将启动 CLI 并引导启动 logger（如果配置了）和 Hive 历史记录文件，并最终显示 Hive CLI 提示：

```
hive>
```

使用以下命令可以随时退出 Hive CLI：

```
hive> exit;
```

通过传递文件名选项 `-f` 和要执行的脚本的路径，Hive 也可以直接从命令行以非交互模式运行：

```
~$ hive -f ~/hadoop-fundamentals/hive/init.hql
~$ hive -f ~/hadoop-fundamentals/hive/top_50_players_by_homeruns.hql >>
~/homeruns.tsv
```

此外，带引号的查询字符串选项 `-e` 让你能从命令行运行内联命令：

```
~$ hive -e 'SHOW DATABASES;'
```

可以使用 `-H` 标志查看 CLI 的 Hive 选项的完整列表：

```
~$ hive -H

usage: hive
 -d,--define <key=value>              Variable substitution to apply to hive
                                          commands. e.g. -d A=B or --define
    --database <databasename>         Specify the database to use
-e <quoted-query-string>              SQL from command line
-f <filename>                         SQL from files
-H,--help                             Print help information
-h <hostname>                         connecting to Hive Server on remote ho
   --hiveconf <property=value>        Use value for given property
   --hivevar <key=value>              Variable substitution to apply to hive
                                           commands. e.g. --hivevar A=B
-i <filename>                         Initialization SQL file
-p <port>                             connecting to Hive Server on port numb
-S,--silent                           Silent mode in interactive shell
-v,--verbose                          Verbose mode (echo executed SQL to the
                                      console)
```

非交互模式为运行已存脚本提供了方便，但是 CLI 使我们能够在 Hive 中轻松地调试和迭代查询。

6.1.2 Hive查询语言

在本节中，我们将通过编写 HQL 语句，创建 Hive 数据库、将 HDFS 中的数据加载到数据库，以及查询数据进行分析。本节引用的数据可以在 GitHub 仓库的 /data 目录中找到。

01. 创建数据库

在 Hive 中创建数据库与在基于 SQL 的 RDBMS 中创建数据库非常相似，使用 `CREATE DATABASE` 或 `CREATE SCHEMA` 语句：

```
hive> CREATE DATABASE log_data;
```

当 Hive 创建新数据库时，模式定义数据存储在 Hive Metastore 中。如果 Metastore 中已经有该数据库，Hive 将抛出错误。我们可以通过使用 `IF NOT EXISTS` 来检查数据库是否存在：

```
hive> CREATE DATABASE IF NOT EXISTS log_data;
```

然后运行 `SHOW DATABASES` 来验证数据库是否已被创建。Hive 将返回在 Metastore 中找到的所有数据库，以及默认的 Hive 数据库：

```
hive> SHOW DATABASES;
OK
default
log_data
Time taken: 0.085 seconds, Fetched: 2 row(s)
```

此外，可以使用 `USE` 命令设置工作数据库：

```
hive> USE log_data;
```

这样就可以在 Hive 中创建一个数据库。可以通过在数据库中创建表定义，描述数据的结构。

02. 创建表

Hive 提供了一个类似 SQL 的 `CREATE TABLE` 语句，最简单的形式由一个表名和一个列定义构成：

```
CREATE TABLE apache_log (
    host STRING,
    identity STRING,
    user STRING,
    time STRING,
    request STRING,
    status STRING,
    size STRING,
    referer STRING,
    agent STRING
);
```

但是由于 Hive 数据存储在文件系统，通常在 HDFS 或本地文件系统中，所以 `CREATE TABLE` 命令还使用可选子句指定行格式，使用 `ROW FORMAT` 子句告诉 Hive 如何读取文件中的每一行并映射到我们的列。例如，可以指明数据位于由制表符分隔字段的文件中：

```
hive> CREATE TABLE shakespeare (
    lineno STRING,
    linetext STRING
)
ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t';
```

Apache 访问日志的每行根据通用日志格式（https://httpd.apache.org/docs/1.3/logs.html#common）进行结构化。好在 Hive 为我们提供了一种方法，能将正则表达式应用于已知格式的记录，从而将每行反序列化或解析为各个组成字段。我们将使用 Hive 的 serializer-deserializer 行格式选项 SERDE 和 RegexSerDe 库来指定反序列化，并将字段映射到表列的正则表达式。需要手动将 lib 文件夹中的 hive-serde JAR 添加到当前 `hive` 会话，以便使用 RegexSerDe 包：

```
hive> ADD JAR /srv/hive/lib/hive-serde-0.13.1.jar;
```

现在删除之前创建的 `apache_log` 表，并使用自定义序列化器重新创建它：

```
hive> DROP TABLE apache_log;

hive> CREATE TABLE apache_log (
    host STRING,
    identity STRING,
    user STRING,
    time STRING,
    request STRING,
    status STRING,
    size STRING,
    referer STRING,
    agent STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES ("input.regex" = "([^ ]*) ([^ ]*) ([^ ]*) (-|\\
*\\])([^ \"]*|\"[^\"]*\") (-|[0-9]*) (-|[0-9]*)(?: ([^ \"]*|\".*\")
*|\".*\"))?", "output.format.string" = "%1$s %2$s %3$s %4$s %5$s %6$
```

```
%8$s %9$s")
STORED AS TEXTFILE;
```

创建之后就可以使用 DESCRIBE 来验证表定义：

```
hive> DESCRIBE apache_log;
OK
host                    string                  from deserializer
identity                string                  from deserializer
user                    string                  from deserializer
time                    string                  from deserializer
request                 string                  from deserializer
status                  string                  from deserializer
size                    string                  from deserializer
referrer                string                  from deserializer
agent                   string                  from deserializer
Time taken: 0.553 seconds, Fetched: 9 row(s)
```

请注意，在这个表中，所有列都使用 Hive 原始数据类型 string 定义。Hive 支持 SQL 用户熟悉的许多其他原始数据类型，并且通常与 Java 支持的原始类型（https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL）对应。表 6-1 列举了这些原始数据类型。

表6-1：Hive的原始数据类型

类型	描述	示例
TINYINT	8 位带符号整数，从 -128 到 127	127
SMALLINT	16 位带符号整数，从 -32 768 到 32 767	32767
INT	32 位带符号整数	2147483647
BIGINT	64 位带符号整数	9223372036854775807
FLOAT	32 位单精度浮点数	1.99
DOUBLE	64 位双精度浮点数	3.14159265359
BOOLEAN	true 或 false	true
STRING	最大 2GB 的字符串	hello world
TIMESTAMP	纳秒级时间戳	1400561325

除原始数据类型之外，Hive 还支持可以存储值集合的复杂数据类型，表 6-2 列举了这些类型。

表6-2：Hive的复杂数据类型

类型	描述	示例
ARRAY	有序集合，数组中元素的类型必须相同	recipients ARRAY<email:STRING>
MAP	无序键值对集合，键必须是原始数据类型，但值可以是任意类型	files MAP<filename:STRING,size:INT>
STRUCT	任意类型元素集合	address STRUCT<street:STRING,city:STRING, state: STRING,zip:INT>

这乍看之下可能很别扭，因为关系数据库通常不支持集合类型，而是将相关集合存储在单独的表中，以维持第一范式，最小化数据重复和数据不一致的风险。然而，在像 Hive 这样的通过顺序扫描磁盘来处理大量非结构化数据的大数据系统中，读取嵌入集合能大大提高检索性能。2

有关 Hive 支持的表和数据类型选项的完整介绍，请参见 Apache Hive 语言手册（https://cwiki.apache.org/confluence/display/Hive/LanguageManual）。

03. 加载数据

创建表和定义模式之后，就可以将数据加载到 Hive 了。请注意 Hive 和传统 RDBMS 在强化模式（schema enforcement）上有一个重要区别：Hive 不会对数据执行任何证明它是否符合表模式的验证，也不会在将数据加载到表中时执行任何转换。

传统的关系数据库通过拒绝写入不符合模式定义的数据，强制执行写时模式（schema on write）；而 Hive 对查询只强制执行读时模式（schema on read）。在读取数据文件时，如果文件结构与定义的模式不匹配，Hive 通常会为缺失的或类型不匹配的字段返回 `null` 值，并尝试从错误中恢复。读时模式初始加载的速度非常快，因为数据不以数据库的内部格式读取、解析和序列化到磁盘。加载操作纯粹是将数据文件移动到 Hive 表（https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML）中对应位置的复制 / 移动操作。

Hive 中的数据加载通过 `LOAD DATA` 命令批量完成，也可以使用 `INSERT` 命令插入另一个查询的结果完成。首先，将 Apache 日志数据文件（http://ita.ee.lbl.gov/html/contrib/Calgary-HTTP.html）复制到 HDFS，然后将其加载到之前创建的表中：

```
~$ hadoop fs -mkdir statistics
~$ hadoop fs -mkdir statistics/log_data
~$ hadoop fs -copyFromLocal ~/hadoop-fundamentals/data/log_data/apac
```

可以使用 tail 命令验证 apache.log 是否成功上传到了 HDFS：

```
~$ hadoop fs -tail statistics/log_data/apache.log
```

一旦文件上传到了 HDFS，就返回 Hive CLI 并使用 `log_data` 数据库：

```
~$ $HIVE_HOME/bin/hive

hive> use log_data;
OK
Time taken: 0.221 seconds
```

使用 `LOAD DATA` 命令，并指定日志文件的 HDFS 路径，将内容写入到 `apache_log` 表中：

```
hive> LOAD DATA INPATH 'statistics/log-data/apache.log'
OVERWRITE INTO TABLE apache_log;

Loading data to table log_data.apache_log
rmr: DEPRECATED: Please use 'rm -r' instead.
Deleted hdfs://localhost:9000/user/hive/warehouse/log_data.db/apache
Table log_data.apache_log stats: [numFiles=1, numRows=0, totalSize=5
rawDataSize=0]
OK
Time taken: 0.902 seconds
```

`LOAD DATA` 是 Hive 的批量加载命令。`INPATH` 携带一个指向默认文件系统（本例中为 HDFS）中的路径的参数。我们还可以使用 `LOCAL INPATH` 来指定本地文件系统上的路径。Hive 将文件移动到仓库位置。如果使用 `OVERWRITE` 关键字，则目标表中的所有已有数据将被删除并由数据文件输入替换；否则，新数据将被添加到表中。

数据复制并加载后，Hive 输出了一些关于加载数据的统计信息；虽然报告的 `num_rows` 为 0，但你可以通过运行 `SELECT COUNT` 来验证实际行数（省略输出）：

```
hive> SELECT COUNT(1) FROM apache_log;
Total MapReduce jobs = 1
Launching Job 1 out of 1
...
OK
726739
Time taken: 34.666 seconds, Fetched: 1 row(s)
```

可以看到，这个 Hive 查询运行时，实际上执行了一个 MapReduce 作业来执行聚合。在 MapReduce 作业执行后，你可以看到 apache_log 表目前有 726 739 行。

2《Hive 编程指南》，Capriolo 等人著。

6.1.3 Hive数据分析

你已经定义了一个模式并将数据加载到了 Hive 中，现在就可以对 Hive 数据库运行 HQL 查询，从而对数据进行实际的数据分析了。在本节中，我们将编写和运行 HQL 查询，根据先前导入的 Apache 访问日志数据，确定远程流量访问的高峰月份。

01. 分组

在上一节中，我们将一份 Apache 访问日志文件加载到了名为 apache_log 的 Hive 表中，其中包含 Apache Common Log 格式（http://httpd.apache.org/docs/2.2/logs.html#accesslog）的 Web 日志数据：

```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif
2326
```

考虑一个计算每个自然月访问数的 MapReduce 程序。尽管这是一个非常简单的分组计数问题，但是要实现这个 MapReduce 程序仍然需要耗费不少的精力——除了要编写 mapper、reducer 和配置作业的 main 函数之外，还要编译和创建 JAR 文件。但有了 Hive 的话，这个问题将与运行 SQL 的 GROUP BY 查询一样简单直观：

```
hive> SELECT
        month,
        count(1) AS count
```

```
      FROM (SELECT split(time, '/')[1] AS month FROM apache_log) l
      GROUP BY month
      ORDER BY count DESC;
OK
Mar 99717
Sep 89083
Feb 72088
Aug 66058
Apr 64984
May 63753
Jul 54920
Jun 53682
Oct 45892
Jan 43635
Nov 41235
Dec 29789
NULL    1903
Time taken: 84.77 seconds, Fetched: 13 row(s)
```

Hive 查询和 MapReduce 程序都要对输入进行分词，并提取月份令牌作为聚合字段。不仅如此，Hive 还提供了简洁自然的查询接口来执行分组；又因为数据被结构化为 Hive 表，所以我们可以轻松地在其他任何字段上执行其他即席查询：

```
hive> SELECT host, count(1) AS count FROM apache_log GROUP BY host
ORDER BY count;
```

除了计数之外，Hive 还支持其他聚合函数来计算数字列的总和、平均值、最小值、最大值以及方差、标准差和协方差等统计聚合。使用这些内置聚合函数时，可以通过将以下属性设置为 `true` 来提高聚合查询的性能：

```
hive> SET hive.map.aggr = true;
```

这种设置告诉 Hive 在 map 阶段执行"顶层"（top-level）聚合，这与执行 `GROUP BY` 后再进行聚合不同。但请注意，此设置将需要更多内存。**3** 在 Hive 文档的"Hive Operators and User- Defined Functions（UDFs）"（http://bit.ly/1r1RnGC）中可以找到内置聚合函数的完整列表。

Hive 还提供了轻松存储计算结果的捷径。你可以创建新的表来存储这些查询返回的结果，以便进行记录保存和分析：

```
hive> CREATE TABLE
        remote_hits_by_month
        AS
    SELECT
        month,
        count(1) AS count
    FROM (
        SELECT split(time, '/')[1] AS month
        FROM apache_log
        WHERE host == 'remote'
        ) l
    GROUP BY month
    ORDER BY count DESC;
```

`CREATE TABLE AS SELECT`（CTAS）操作非常有用，它能从现有 Hive 表过滤和聚合，从而派生并构建新表。

02. 聚合和连接

当在单个结构化的数据集中查询和聚合数据时，Hive 能提供一些便利，我们对此也有所涉及。但在多个数据集之上执行更复杂的聚合时，Hive 才能真正物尽其用。

在第 3 章中，我们开发了一个 MapReduce 程序，根据交通研究与创新技术管理局（RITA，http://1.usa.gov/1r1RJ09）收集的航班数据分析美国航空公司的准点情况。那一章将该准点数据集进行了归一化，让单个数据文件包含所有必需的数据；但事实上，从 RITA 网站下载的数据包括航空公司和飞机的代码，它们必须分别参照单独的查找数据集。2014 年 4 月的数据已被放入 GitHub 仓库的 data/flight_data 目录。

ontime_flights.tsv 中的准点航班数据的每一行都包含一个表示航空公司代码（如 19805）的整数值和一个表示飞机代码（如“AA”）的字符串值。航空公司代码可以与 airlines.tsv 文件中相应的代码进行连接，该文件每行包含代码和相应的描述：

```
19805    American Airlines Inc.: AA
```

同理，飞机代码可以与 carriers.tsv 中对应的代码进行连接，该文件包含代码、相应的航空公司名称和生效日期：

```

```

```
AA    American Airlines Inc. (1960 - )
```

要在 MapReduce 程序中实现这些连接，需要在 map 端加载查找表到内存中进行连接，或者在 reduce 端连接。这两种方法都需要耗费大量精力编写配置作业的 MapReduce 代码；但是通过 Hive，则可以简单地将这些附加的查找数据集加载到单独的表中，并在 SQL 查询中执行连接。

假设我们已经将数据文件上传到了 HDFS 或本地文件系统。首先，为航班数据创建一个新的数据库：

```
hive> CREATE DATABASE flight_data;
OK
Time taken: 0.741 seconds
```

然后，为准点数据和查找表定义模式并加载数据（出于可读性考虑，省略输出和添加换行）：

```
hive> CREATE TABLE flights (
        flight_date DATE,
        airline_code INT,
        carrier_code STRING,
        origin STRING,
        dest STRING,
        depart_time INT,
        depart_delta INT,
        depart_delay INT,
        arrive_time INT,
        arrive_delta INT,
        arrive_delay INT,
        is_cancelled BOOLEAN,
        cancellation_code STRING,
        distance INT,
        carrier_delay INT,
        weather_delay INT,
        nas_delay INT,
        security_delay INT,
        late_aircraft_delay INT
    )
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t'
    STORED AS TEXTFILE;

hive> CREATE TABLE airlines (
        code INT,
        description STRING
```

```
    )
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t'
    STORED AS TEXTFILE;

hive> CREATE TABLE carriers (
        code STRING,
        description STRING
    )
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t'
    STORED AS TEXTFILE;

hive> CREATE TABLE cancellation_reasons (
        code STRING,
        description STRING
    )
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\t'
    STORED AS TEXTFILE;

hive> LOAD DATA LOCAL INPATH
        '${env:HOME}/hadoop-fundamentals/data/flight_data/ontime_fli
      OVERWRITE INTO TABLE flights;

hive> LOAD DATA LOCAL INPATH
        '${env:HOME}/hadoop-fundamentals/data/flight_data/airlines.t

OVERWRITE INTO TABLE airlines;

hive> LOAD DATA LOCAL INPATH
        '${env:HOME}/hadoop-fundamentals/data/flight_data/carriers.t
      OVERWRITE INTO TABLE carriers;

hive> LOAD DATA LOCAL INPATH
        '${env:HOME}/hadoop-fundamentals/data/flight_data/
        cancellation_reasons.tsv'
      OVERWRITE INTO TABLE cancellation_reasons;
```

要获取航空公司及其各自的平均起飞延误时间列表，只需要基于航空公司代码对航班和航空公司执行 SQL `JOIN`，然后使用聚合函数 `AVG()` 计算按航空公司描述分组的平均 `depart_delay`：

```
hive> SELECT
        a.description,
        AVG(f.depart_delay)
      FROM airlines a
      JOIN flights f ON a.code = f.airline_code
      GROUP BY a.description;
```

```
AirTran Airways Corporation: FL 8.035840978593273
Alaska Airlines Inc.: AS    4.746143501305276
American Airlines Inc.: AA  10.085038790027395
American Eagle Airlines Inc.: MQ    11.048787878787879
Delta Air Lines Inc.: DL    8.149843785719728
ExpressJet Airlines Inc.: EV    15.762459814292642
Frontier Airlines Inc.: F9  12.319591084296967
Hawaiian Airlines Inc.: HA  2.872051586628203
JetBlue Airways: B6 12.090553084509766
SkyWest Airlines Inc.: OO   10.086447897294379
Southwest Airlines Co.: WN  14.722817981677437
US Airways Inc.: US 7.363223345079652
United Air Lines Inc.: UA   11.124291343587137
Virgin America: VX 9.98681228106326
Time taken: 22.786 seconds, Fetched: 14 row(s)
```

如你所见，与在 MapReduce 中执行连接相比，在 Hive 中执行连接可以显著减少编码工作量。更重要的是，我们定义的结构化 Hive 数据模式使我们能够轻松添加或更改查询。让我们来修改查询，返回按飞机分组的平均起飞延误时间：

```
hive> SELECT
        c.description,
        AVG(f.depart_delay)
      FROM carriers c
      JOIN flights f ON c.code = f.carrier_code
      GROUP BY c.description;

Aces Airlines (1992 - 2003) 9.98681228106326
AirTran Airways Corporation (1994 - )   8.035840978593273
Alaska Airlines Inc. (1960 - )  4.746143501305276
American Airlines Inc. (1960 - )    10.085038790027395
American Eagle Airlines Inc. (1998 - )  11.048787878787879
Atlantic Southeast Airlines (1993 - 2011)   15.762459814292642
Delta Air Lines Inc. (1960 - )  8.149843785719728
ExpressJet Airlines Inc. (2012 - )  15.762459814292642
Frontier Airlines Inc. (1960 - 1986)    8.035840978593273
Frontier Airlines Inc. (1994 - )    12.319591084296967
Hawaiian Airlines Inc. (1960 - )    2.872051586628203
JetBlue Airways (2000 - )   12.090553084509766
Simmons Airlines (1991 - 1998)  11.048787878787879
SkyWest Airlines Inc. (2003 - ) 10.086447897294379
Southwest Airlines Co. (1979 - )    14.722817981677437
US Airways Inc. (1997 - )   7.363223345079652
USAir (1988 - 1997) 7.363223345079652
United Air Lines Inc. (1960 - ) 11.124291343587137
Virgin America (2007 - )    9.98681228106326
Time taken: 22.76 seconds, Fetched: 19 row(s)
```

> Hive 可能比较适用于这些使用场景：使用的数据集的格式是结构化的、基于表格的；要进行的计算是面向批处理的 OLAP 查询，而不是实时的、面向行的 OLTP 事务。如果想了解更多有关使用和优化 Hive 的信息，推荐你阅读以示例驱动的《Hive 编程指南》，这是一本非常优秀的图书。

3《Hive 编程指南》，Edward Capriolo、Dean Wampler、Jason Rutherglen 著。

6.2 HBase

在上一节中，我们了解了如何使用 Hive 对存储在 HDFS 中的大型结构化数据集执行基于 SQL 的分析。但我们也发现，虽然 Hive 在 Hadoop 中提供了一个熟悉的数据操作范式，但它并不会改变存储和处理模式，而是仍然以批处理方式使用 HDFS 和 MapReduce。

回想一下，由于 HDFS 被设计为写一次、读多次（WORM）的文件系统，因此它针对顺序读取进行了优化，处理起需要对数据进行频繁或快速的行级更新的用例效率低下。这种数据访问模式通常被称为“随机访问”，需要采用这种实时、低延迟读 / 写访问的应用程序也越来越多。以呈爆炸式增长的实时传感器和遥测应用程序为例，如 NOAA（https://www.ncdc.noaa.gov/data-access/land-based-station-data）使用的从远程观测站收集天气数据的应用程序或 NASA 用于记录来自无人驾驶飞船的数据传输的深空网络（https://solarsystem.nasa.gov/basics/bsf18-1.php）。这些应用程序必须存储和处理多个传输设备以极快的速率传输过来的大量事件数据，并在查询数据时确保数据的正确性或一致性。因此，对于需要对数据进行随机、实时读 / 写访问的用例，需要在标准的 MapReduce 和 Hive 之外寻找数据持久层和处理层技术。

对许多数据分析应用程序来说，使用传统的关系方法建模还存在挑战。像 Facebook 的实时分析应用程序“Insights for Websites”平台（每秒跟踪超过 20 万个事件 4）和 StumbleUpon（http://www.stumbleupon.com）的实时推荐系统 5，它们需要同时记录来自许多数据源的大量数据事件。这些类型的实时应用程序需要记录大量基于时间的事件，这些事件往往有许多种可能的结构。数据可能以某个特定值为键（如用户），但值通常表示为任意元数据的集合。以“Like”和“Share”两个事件为例，它们需要不同的列值，如表 6-3 所示。

4Alex Himel,“Building Realtime Insights” (https://www.facebook.com/note.php?note_id=10150103900258920), Facebook Engineering note, 2011.03.05.

5Katie Gray,“Why We Love HBase”(http://www.stumbleupon.com/blog/why-we-love-hbase/), StumbleUpon Official Blog, 2010.11.18.

表6-3：非结构化事件

			[illegible]			
[illegible]						
[illegible]						
[illegible]						

这些类型的数据应用程序有存储稀疏数据的需求。在关系模型中，行是稀疏的，但列不是；也就是说，在将新行插入表后，数据库将为每个列分配存储空间，不管该字段是否有值。然而，在数据被表示为任意字段或稀疏列的集合的应用程序中，每行可能只使用可用列的一部分，这让标准关系模式既浪费资源又别扭。

6.2.1　NoSQL与列式数据库

如今，许多现代应用程序都面临着规模和敏捷的挑战，NoSQL 数据库也因此应运而生。NoSQL 是一个广泛的概念，通常指非关系数据库，涵盖广泛的数据存储模型，包括图形数据库、文档数据库、键 / 值数据存储和列族数据库。

HBase 被归类为列族或列式数据库，模型建立在 Google 的 BigTable 架构（http://research.google.com/archive/bigtable.html）之上。这种架构让 HBase 具有如下特性：

- 随机（行级）读 / 写访问；
- 强大的一致性；
- “无模式”或灵活的数据建模。

HBase 处理数据建模的方式引入了无模式的特点，它与关系数据库处理数据建模的方式非常不同。HBase 将数据组织到包含行的表中。在表中，行由唯一的行键标识，行键没有数据类型，而是作为字节数组被存储和处理。行键与关系数据库中主键的概念类似，都被自动编入索引。在 HBase 中，表行按照行键进行排序；因为行键是字节数组，所以字符串、long 的二进制表示，乃至序列化的数据结构等几乎一切

都可以作为行键。HBase 将其数据存储为键值对，所有表查找都是通过表的行键或存储记录数据的唯一标识符执行的。

一行中的数据被分组成列族，它们由相关列组成。你可以画出一个 HBase 表，让它包含给定人口的人口普查数据，其中每行代表一个人，通过唯一的 ID 行键进行访问；每行包含个人数据和人口信息的列族，个人数据的列族包含姓名列和地址列，人口信息的列族包含出生日期列和性别列。该示例如图 6-1 所示。

PERSON表

行键	personal_data		demographic		...
PersonID	Name	Address	BirthDate	Gender	...
1	H. Houdini	Budapest, Hungary	1926-10-31	M	
2	D. Copper	New Jersey, USA	1956-09-16	M	
3	Merlin	Stonehenge, England	1136-12-03	F	
...	...	...	...	...	
500,000,000	F. Cadillac	Nevada, USA	1964-01-07	M	

图 **6-1**：**HBase** 模式的人口普查数据

数据仓库和分析数据库的聚合都是在大量数据上进行的，这些数据可能是稀疏的，即不是所有列值都存在，因而按列存储数据比按行存储的优势更明显。尽管列族非常灵活，但列族在实践中并不完全是无模式的。在可以开始将数据插入特定行和列之前，列族就已经被定义了，因为它们会影响 HBase 存储数据的物理格局。[6] 然而，可以根据需要确定和创建组成行的实际列。事实上，每行可以包含不同的列。图 6-2 展示了一个有两行的 HBASE 表，第一个行键使用了三列族，第二个行键仅使用一列。

> 6Amandeep Khurana,“Introduction to HBase Schema Design”; login:, October 2012, Volume 37, Number 5.

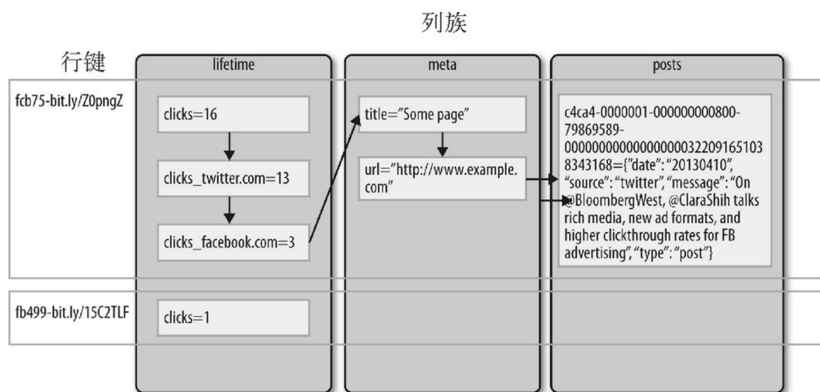


图 **6-2**：列稀疏的社交媒体事件

HBase 和基于 BigTable 的列式数据库还有一个有趣的特征，那就是表的单元格或行、列坐标的交集由时间戳进行版本控制；时间戳存储为一个自 1970 年 1 月 1 日 UTC 以来的长整数，用毫秒表示。因此，HBase 也被描述为一个多维的 map，其中时间提供第三维度，如图 6-3 所示。时间维度以递减顺序索引，因此从 HBase 读取时会先找到最新的值。单元格的内容可以由 `{rowkey, column, timestamp}` 元组引用，或者可以按时间范围扫描一系列单元格值。

t1
t2
t3

key	cf1:colA	cf1:colB	cf2:colC	cf2:ColD
1	ca1-t3		cc1-t2	cd1-t3
2	ca2-t3	cb2-t3		cd2-t3
3	ca3-t3		cc3-t3	

图 **6-3**：**HBase** 时间戳版本

介绍完 HBase 模式设计的主要特性，下面开始学习如何设计和查询一个简单的 HBase 表，用于假设的实时链接分享应用程序。此处假设你已经在开发环境中安装并配置了 HBase，安装和配置 HBase 的步骤见附录 B。

6.2.2　HBase实时分析

可以使用 HBase Shell 或 Java API（使用 HBaseAdmin 接口类，https://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HBaseAdmin.html）创建和更新 HBase 模式。此外，HBase 还支持其他许多客户端，可用于支持非 Java 的编程语言，比如 REST API 接口、Thrift 和 Avro。[7] 这些客户端是包装了原生 Java API 的代理。

7参见 Apache HBase Reference Guide 中的“Apache HBase Book External APIs”（http://hbase.apache.org/book.html#external_apis）。

为了概述 HBase，我们定义并使用 HBase shell 设计了一个链接共享跟踪器，用来跟踪一个链接被共享的次数。但在现实情况中，你会使用原生 Java API 或受支持的客户端库编写应用程序。如果要创建外部网关客户端，请仔细考虑你的使用场景。需要高吞吐量的应用程序与纯粹的二进制格式（如 Thrift 或 Avro）更配，而 REST API 可能更适合低频率的大请求。

01. 生成模式

在 HBase 中设计模式时，要着重考虑数据模型的列族结构以及它对数据访问模式的影响。定义传统关系数据库的模式主要是要准确表示实体和实体间的关系，以及考虑连接和索引等方面的性能，但成功的 HBase 模式定义往往取决于应用程序的预期用例。此外，由于 HBase 不支持连接并且只提供一个索引的行键，所以必须要小心，确保模式可以完全支持所有用例。这通常涉及使用嵌套实体的去规范化和数据重复。

幸运的是，由于 HBase 支持在运行时进行动态列定义，所以即便在创建表之后，也依然有很大的灵活性来修改和扩展模式。

02. 命名空间、表和列族

那么模式的哪些方面值得仔细考虑呢？首先，需要在定义表时声明表名和至少一个列族名；还可以声明自己的可选命名空间（从 Apache HBase v0.96.0 起被支持）作为表的逻辑分组，与关系数据库系统中的数据库类似。8 如果没有声明命名空间，HBase 将使用 `default` 命名空间：

```
hbase> create 'linkshare', 'link'
0 row(s) in 1.5740 seconds
```

我们刚刚在 `default` 命名空间中创建了一个名为 `linkshare` 的表，它有一个名为 `link` 的列族。要想在创建表后再更改表，例如更改或添加列族，需要先禁用该表，以便客户端在 `alter` 操作期间无法访问该表：

```
hbase> disable 'linkshare'
0 row(s) in 1.1340 seconds

hbase> alter 'linkshare', 'statistics'
Updating all regions with the new schema...
1/1 regions updated.
```

```
Done.
0 row(s) in 1.1630 seconds
```

然后，使用 `enable` 命令重新启用该表：

```
hbase> enable 'linkshare'
0 row(s) in 1.1930 seconds
```

使用 `describe` 命令验证该表包含两个带有默认配置的预期列族：

```
hbase> describe 'linkshare'

Table linkshare is ENABLED
COLUMN FAMILIES DESCRIPTION
{NAME => 'link', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW
REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '1',
TTL => 'FOREVER', MIN_VERSIONS => '0', KEEP_DELETED_CELLS => 'FALSE
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
{NAME => 'statistics', DATA_BLOCK_ENCODING => 'NONE',
BLOOMFILTER => 'ROW', REPLICATION_SCOPE => '0', VERSIONS => '1',
COMPRESSION => 'NONE', MIN_VERSIONS => '0', TTL => 'FOREVER',
KEEP_DELETED_CELLS => 'FALSE', BLOCKSIZE => '65536',
IN_MEMORY => 'false', BLOCKCACHE => 'true'}
2 row(s) in 0.1290 seconds
```

这样就创建了一个有两个列族（`link` 和 `statistics`）的 HBase 表（`linkshare`），但是这个表还不包含任何行。在插入行数据之前，需要确定如何设计行键。

03. 行键

良好的行键设计不仅会影响查询表的方式，也会影响数据访问的性能和复杂性。默认情况下，HBase 根据行键按顺序存储行，因此类似的键存储在同一个 RegionServer 中。虽然这样可以更快地进行扫描范围，但在读 / 写操作期间，它也可能导致个别服务器的负载不均匀（称为“RegionServer hotspotting”）。因此，除了要实现我们的数据访问用例之外，还需要考虑各个 region 之间的行键分布。

以当前示例为例，假设我们使用唯一的反向链接 URL 作为行键。强烈推荐你阅读“Apache HBase Reference Guide”中

的“HBase and Schema Design”（http://hbase.apache.org/0.94/book/schema.html），了解优秀的行键设计案例。

04. 使用**put**插入数据

现在这个表可以存储数据了——我们想在 linkshare 应用程序中存储关于链接的描述性数据（例如其标题），同时维护一个跟踪链接共享次数的频率计数器。

我们可以在指定的表 / 行 / 列和可选时间戳坐标的单元格中插入或 put 一个值。要将一个单元格值放入 linkshare 表、行键为 org.hbase.www 的行、link 列族下的当前时间戳的 title 列，可以执行以下操作：

```
hbase> put 'linkshare', 'org.hbase.www', 'link:title', 'Apache HBase
hbase> put 'linkshare', 'org.hadoop.www', 'link:title', 'Apache Hadc
hbase> put 'linkshare', 'com.oreilly.www', 'link:title', 'O\'Reilly.
```

put 操作在插入单个单元格的值时非常有用，而对于增加频率计数器的值，HBase 提供了一种将列作为计数器来处理的特殊机制。否则，在负载较重的情况下，我们可能会遇到针对这些行的激烈竞争，因为我们需要锁定行，读取值，增加值，写回，最后再解锁该行，让其他写入过程能够访问该单元格。**9**

使用 incr 命令，而不是 put 来增加计数器的值：

```
hbase> incr 'linkshare', 'org.hbase.www', 'statistics:share', 1
(COUNTER VALUE is now 1)

hbase> incr 'linkshare', 'org.hbase.www', 'statistics:like', 1
(COUNTER VALUE is now 1)
```

最后一个被传递的选项是增量值，在这个例子中为 1。增加计数器的值将返回更新后的计数器值，但你也可以随时使用 get_counter 命令访问计数器的当前值，指定表名、行键和列即可：

```
hbase> incr 'linkshare', 'org.hbase.www', 'statistics:share', 1
(COUNTER VALUE is now 2)

hbase> get_counter 'linkshare', 'org.hbase.www', 'statistics:share',
```

```
COUNTER VALUE = 2
```

get_counter 方法用于解码计数器的字节数组值，并返回整数值。不幸的是，最新版本的 HBase 在获取计数器值的 shell 命令中引入了一个 bug——该命令的第 4 个参数没有作用。因此，需要传递哑值作为第 4 个参数：

```
hbase> get_counter 'linkshare', 'org.hbase.www', 'statistics:share',
COUNTER VALUE = 2
```

HBase 提供了从表中检索数据的两种通用方法：get 命令通过行键执行查找，从而检索特定行的属性；scan 命令接受一组过滤器规范，并根据指定的规范进行多行迭代。

05. 获取行或单元格值

最简单的形式是，get 命令接受表名与行键，返回行中所有列的最新版本的时间戳和单元格值：

```
hbase> get 'linkshare', 'org.hbase.www'

COLUMN                          CELL
 link:title                     timestamp=1422145743298, value=Apache
 statistics:like                timestamp=1422153344211,
     value=\x00\x00\x00\x00\x00\x00\x00\x1F
 statistics:share               timestamp=1422153337498,
     value=\x00\x00\x00\x00\x00\x00\x00\x02
3 row(s) in 0.0310 seconds
```

请注意，statistics:share 列返回它的字节数组表示的值，并将每个字节打印为十六进制值。要显示计数器值的整数表示，可以使用上一节中提到的 get_counter 命令。

get 命令还接受可选的参数字典，用来指定要检索的单元格值的列、时间戳、时间范围（timerange）和版本。例如，可以指定感兴趣的列：

```
hbase> get 'linkshare', 'org.hbase.www', {COLUMN => 'link:title'}
hbase> get 'linkshare', 'org.hbase.www', {COLUMN => ['link:title',
```

还有一种在 get 中指定列参数的快捷方式，就是在行键之后加上以逗号分隔的列名称：

```
hbase> get 'linkshare', 'org.hbase.www', 'link:title'
hbase> get 'linkshare', 'org.hbase.www', 'link:title', 'statistics:s
hbase> get 'linkshare', 'org.hbase.www', ['link:title', 'statistics:
```

为了指定感兴趣的值的时间范围，传入一个有开始时间戳和结束时间戳（以毫秒为单位）的 TIMERANGE 参数：

```
hbase> get 'linkshare', 'org.hbase.www', {TIMERANGE => [13998877056
```

如果希望获取一定数量的先前版本的单元格值，而不是显式的时间戳范围，可以指定感兴趣的列，并使用 VERSIONS 参数指定要检索的版本数：

```
hbase> get 'linkshare', 'org.hbase.www', {COLUMN => 'statistics:shar
```

虽然这种类型的范围查询似乎对递增为 1 的计数器值意义不大，但它为我们提供了确定分享计数器递增速率的方法，可以用它来确定链接是否为病毒。此外，这些类型的范围过滤器在执行“即时”查询时尤其有用，例如检查在一定时间范围内的指标。

06. 扫描行

scan 操作类似于数据库游标或迭代器，利用底层顺序存储机制，根据 scanner 规格迭代行数据。可以使用 scan 扫描整个 HBase 表或指定范围的行。

scan 的用法和 get 类似，它也接受 COLUMN、TIMESTAMP、TIMERANGE 和 FILTER 参数。但是，你不能指定单个行键，而是要指定一个可选的 STARTROW 和 / 或 STOPROW 参数，将扫描限制在特定的行范围内。如果不提供 STARTROW 和 STOPROW，scan 操作将扫描整个表。

事实上，你可以将表名称作为参数调用 scan，从而显示表的所有内容：

```
hbase> scan 'linkshare'

ROW                     COLUMN+CELL
com.oreilly.www         column=link:title, timestamp=1422153270279,
value=O'Reilly.com
org.hadoop.www          column=link:title, timestamp=1422153262507,
value=Apache Hadoop
org.hbase.www           column=link:title, timestamp=1422145743298,
value=Apache HBase
org.hbase.www           column=statistics:like, timestamp=14221533442l
    value=\x00\x00\x00\x00\x00\x00\x00\x1F
org.hbase.www           column=statistics:share, timestamp=142215333749
    value=\x00\x00\x00\x00\x00\x00\x00\x02
3 row(s) in 0.0290 seconds
```

请记住，HBase 中的行以字典顺序存储。10 例如，1~100 将按如下顺序排序：

```
1,10,100,11,12,13,14,15,16,17,18,19,2,20,21,...,9,91,92,93,94,95,96,
```

扫描 `org.hbase.www` 行之后行的 `link:title` 列：

```
hbase> scan 'linkshare', {COLUMNS => ['link:title'], STARTROW => 'or

ROW                   COLUMN+CELL
 org.hbase.www        column=link:title, timestamp=1453184861236,
 value=Apache HBase
1 row(s) in 0.0250 seconds
```

但是 `STARTROW` 和 `ENDROW` 的值不需要行键的完全匹配。它将匹配大于等于给定起始行且小于等于结束行的第一个行键；因为这些参数是包含端点的（inclusive），所以如果 `ENDROW` 的值与 `STARTROW` 相同，就不需要指定 `ENDROW` 了：

```
hbase> scan 'linkshare', {COLUMNS => ['link:title'], STARTROW => 'or

ROW                    COLUMN+CELL
org.hadoop.www         column=link:title, timestamp=1422153262507,
value=Apache Hadoop
org.hbase.www          column=link:title, timestamp=1422145743298,
value=Apache HBase
2 row(s) in 0.0210 seconds
```

07. 过滤器

HBase 提供了一些过滤器类，可以进一步过滤 `get` 或 `scan` 操作返回的行数据。这些过滤器可以提供更有效的手段来限制 HBase 返回的行数据，并将行过滤操作的负担从客户端转移到服务器。HBase 可用的一些过滤器包括：

RowFilter（http://bit.ly/1r1Y7Ez）

基于行键值进行数据过滤。

ColumnRangeFilter（http://bit.ly/1r1Yb7m）

支持高效的行内扫描，可用于获取非常宽的行的部分列（也就是说，当一行有 100 万列时，你只想查看列 bbbb-bbdd）。

SingleColumnValueFilter（http://bit.ly/1r1Yf70）

基于列值过滤单元格。

RegexStringComparator（http://bit.ly/1r1Ydft）

用于检验给定正则表达式是否与列中的单元格值匹配。

HBase 的 Java API（https://hbase.apache.org/apidocs/）提供了一个 Filter（http://bit.ly/1r1YizN）接口、一个 FilterBase 抽象类（http://bit.ly/1r1YizN），以及一些专用的 Filter 子类（http://bit.ly/1r1YsXP）。也可以通过继承 FilterBase 抽象类并实现关键抽象方法来创建自定义过滤器。

最好在 Java 程序中使用 HBase API 来应用 HBase 过滤器，因为它们通常需要导入多个依赖的过滤器和比较器类，但是我们可以在 shell 中演示一个过滤器的简单示例。

首先，导入必需的类，包括用于将列族、列和值转换为字节的 `org.apache.hadoop.hbase.util.Bytes`、过滤器和比较器类：

```
hbase> import org.apache.hadoop.hbase.util.Bytes
hbase> import org.apache.hadoop.hbase.filter.SingleColumnValueFilter
hbase> import org.apache.hadoop.hbase.filter.BinaryComparator
hbase> import org.apache.hadoop.hbase.filter.CompareFilter
```

```
```

接下来，创建一个过滤器，筛选出 `statistics:like` 列值 ≥ 10 的行：

```
hbase> likeFilter = SingleColumnValueFilter.new(Bytes.toBytes('stati
    Bytes.toBytes('like'),
    CompareFilter::CompareOp.valueOf('GREATER_OR_EQUAL'),
    BinaryComparator.new(Bytes.toBytes(10)))
```

因为不是所有行的该列都有值，所以需要设置一个标志，告诉过滤器跳过该列没有值的行：

```
hbase> likeFilter.setFilterIfMissing(true)
```

此时就可以使用配置好的过滤器运行 `scan` 操作了：

```
hbase> scan 'linkshare', { FILTER => likeFilter }

ROW                     COLUMN+CELL
org.hbase.www           column=link:title, timestamp=1422145743298,
    value=Apache HBase
org.hbase.www           column=statistics:like, timestamp=142215334421
    value=\x00\x00\x00\x00\x00\x00\x00\x1F
org.hbase.www           column=statistics:share, timestamp=14221533374
    value=\x00\x00\x00\x00\x00\x00\x00\x02
1 row(s) in 0.0470 seconds
```

这段代码应该返回 `statistics:like` 的列值≥ 10 的所有行；本示例可能包括行键 `com.oreilly.www`。

08. **HBase的拓展阅读**

HBase 在存储大量结构灵活的流式数据时非常有用，它还能一次查询该数据的一小块，同时确保：

- 数据“完整”（如销售或财务数据）；
- 数据会随时间变化；
- 可以查询和更新单行和部分行和列。

HBase 并不是 RDBMS、HDFS 或者 Hive 的替代品，而是提供了一种方法，在利用 Hadoop 的数据可扩展性的同时，实现对

数据的随机访问。HBase 可以与传统的 SQL 或 Hive 结合，以支持需要的查询快照、范围和聚合数据。

有关使用和集成 HBase 的更多信息，建议你查阅官方的“Apache HBase Reference Guide”（http://hbase.apache.org/book.html）以及 Lars George 所著的《Hbase 权威指南》。

8参见 Apache HBase Reference Guide 中的“Namespace”（http://hbase.apache.org/book.html#namespace）。

9《HBase 权威指南》，Lars George 著。

10参见 Apache HBase Reference Guide 中的“Data Model: Rows”（http://hbase.apache.org/book.html#datamodel）。

6.3　小结

本章介绍了 Hive 和 HBase。许多人把 Hive 当作 Hadoop 中 SQL 查询的事实标准，把 HBase 当作在 Hadoop 之上运行的、最流行的 NoSQL 数据库之一。但在深入了解 Hadoop 数据分析之后，你会发现数据仓储和数据挖掘领域中还有许多 Hadoop 项目和工具值得研究，不过这超出了本书的范围。

除了 Hive 之外，还有其他几个查询引擎也可以对 HDFS 或 HBase 进行 SQL 查询。Impala（http://impala.io）通过执行本地内存计算来提供低延迟查询，从而避免了执行 MapReduce 作业的开销；Spark SQL（https://spark.apache.org/sql/）通过运行查询（如 Spark 作业）也支持高性能 SQL 查询。

Hadoop 的优势在于，它可以灵活地支持和使用许多查询和处理引擎，选择最适合特定用例的工具。有关 Hadoop 其他数据仓储和数据挖掘解决方案（包括其他 SQL-on-Hadoop 项目）的更多信息，请参见 Mark Grover、Ted Malaska、Jonathan Seidman 和 Gwen Shapira 合著的《Hadoop 应用架构》。

第 7 章　数据采集

Hadoop 最突出的一个优点在于它天生就是无模式的。只要实现 Hadoop 的 Writable 接口或 DBWritable 接口并编写正确解析数据的 MapReduce 代码，Hadoop 就可以使用任何来源、任何类型或任何格式的数据，不管其结构如何（或是否缺少结构）。然而，如果输入数据驻留在关系数据库中且已结构化，使用数据已知的模式可以更方便地将数据导入 Hadoop；这种方法也比将 CSV 上传到 HDFS 并手动解析更高效。

Sqoop 用于在关系数据库管理系统（relational database management system，RDBMS）和 Hadoop 之间传输数据。它依靠 RDBMS 为要导入的数据提供模式描述，让大部分数据转换过程自动化。我们将在本章中看到，对于将关系数据库作为主要或中间数据存储的数据基础设施来说，Sqoop 是分析流程中一个非常有用的环节。

虽然用 Sqoop 将已经驻留在关系数据库中的数据大批量加载到 Hadoop 中并无不妥，但许多新的应用程序和系统涉及快速流动的数据，如应用程序日志、GPS 追踪、社交媒体动态和传感器数据。我们需要将这些数据直接加载到 HDFS 中，从而在 Hadoop 中进行处理。为了应对高吞吐量并处理由这些系统生成的基于事件的数据，就需要支持从多个源持续采集数据到 Hadoop 中。

Apache Flume 旨在从大量不同来源高效地采集、聚合和移动大量日志数据到集中的数据存储中。虽然 Flume 通常用于将流式日志数据引入 Hadoop（通常是 HDFS 或 HBase），但实际上 Flume 的数据源非常灵活，可以通过自定义向它兼容的任何消费者传输各种类型的事件数据，包括网络流量数据、社交媒体生成的数据和传感器数据。本章将介绍如何使用 Flume 从自定义日志中采集流式数据并将其传输到 Hadoop。

7.1　使用Sqoop导入关系数据

Sqoop（SQL-to-Hadoop）是由 Cloudera[1] 创建的关系数据库导入 / 导出工具，现在是 Apache 顶级项目。Sqoop 的设计初衷是在关系数据库（例如 MySQL 或 Oracle）和 Hadoop 数据存储（例如 HDFS、Hive 和 HBase）之间传输数据。它通过直接从 RDBMS 读取模式信息，自动执行大部分数据传输过程，然后使用 MapReduce 将数据导入和导出 Hadoop。[2]

1Aaron Kimball,“Cloudera—Introducing Sqoop”(https://blog.cloudera.com/blog/2009/06/introducing-sqoop/),Cloudera Engineering Blog, June 1, 2009.

2参见 Sqoop User Guide（http://sqoop.apache.org/docs/1.4.6/SqoopUserGuide.html）。

Sqoop 在将数据维持在生产状态的同时，将其复制到 Hadoop 中，从而进行进一步分析，避免修改生产数据库。我们将介绍几种使用 Sqoop 将数据从 MySQL 数据库导入 Hadoop 数据存储（例如 HDFS、Hive 和 HBase）的方法。

> 本章的 Sqoop 示例假设 MySQL 数据库安装在 Sqoop 所在的主机上且可通过 localhost 访问。要安装和配置本地 MySQL 数据库，请遵循 MySQL 网站上的官方安装指南（https://dev.mysql.com/doc/mysql-apt-repo-quick-guide/en/）或 Linode 网站上的简明指南（https://www.linode.com/docs/databases/mysql/install-mysql-on-ubuntu-14-04/）。请记住，大多数命令需要使用 `sudo`；而且，不要为 servername 设置主机名，因为这会在尝试通过 localhost 连接时发生冲突。

本章假设你已经安装了与你的 Hadoop 版本兼容的 Sqoop 最新稳定版本，将 Hadoop 配置为了伪分布式模式，并且所有 HDFS 和 YARN 进程也在运行中。本章将使用 MySQL 作为示例的源和目标 RDBMS，因此还假设 MySQL 数据库与 Hadoop/Sqoop 服务位于相同的主机上，可通过 localhost 和默认端口 3306 访问。安装 Sqoop 并配置 MySQL 的步骤可以在附录 B 中找到。

7.1.1　从MySQL导入HDFS

从关系数据库（如 MySQL）导入数据时，Sqoop 会从源数据库读取导入数据所需的元数据，然后提交一个仅有 map 的 Hadoop 作业，根据上一步捕获的元数据，传输实际的表数据。该作业会生成一组序列化文件，比如以符号分隔的文本文件、二进制格式（例如 Avro）文件，或包含导入的表或数据集副本的 SequenceFile 文件。3 默认情况下，文件以逗号分隔，并保存在 HDFS 目录中，目录名称与源表名称相对应。我们将使用这些默认设置将数据从 MySQL 导入 HDFS。

3参见 Sqoop User Guide 中的“Basic Usage”（http://bit.ly/1r23vHN）。

假设 MySQL 已经安装好了，继续下一步，创建一个有一些表和数据的示例数据库。首先创建一个名为 energydata 的数据库和一个名为 average_price_by_state 的表：

```
~$ mysql -uroot -p

mysql> CREATE DATABASE energydata;

Query OK, 1 row affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON energydata.* TO '%'@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> GRANT ALL PRIVILEGES ON energydata.* TO ''@'localhost';
Query OK, 0 rows affected (0.00 sec)

mysql> USE energydata;

mysql> CREATE TABLE average_price_by_state(
    year INT NOT NULL,
    state VARCHAR(5) NOT NULL,
    sector VARCHAR(255),
    residential DECIMAL(10,2),
    commercial DECIMAL(10,2),
    industrial DECIMAL(10,2),
    transportation DECIMAL(10,2),
    other DECIMAL(10,2),
    total DECIMAL(10,2)
);
Query OK, 0 rows affected (0.02 sec)

mysql> quit;
```

加载到 average_price_by_state 表中的数据由美国能源信息署（http://www.eia.gov/electricity/data/state/）提供，这些数据是 1990 年至 2012 年期间各年各州和各供应商每千瓦时（KwH）的平均电价。你可以在本书 GitHub 仓库的 /data 目录中找到名为 avgprice_kwh_state.zip 的压缩文件，其中包含名为 avgprice_kwh_state.csv 的 CSV 文件。下载该文件并将其加载到刚刚创建的 MySQL 表中：

```
~$ mysql -h localhost -u root -p energydata --local-infile=1

mysql> LOAD DATA LOCAL INFILE
    '/home/hadoop/hadoop-fundamentals/data/avgprice_kwh_state.csv'
    INTO TABLE average_price_by_state
    FIELDS TERMINATED BY ','
    LINES TERMINATED  BY '\n' IGNORE 1 LINES;
```

```
Query OK, 3272 rows affected, 6 warnings (0.03 sec)
Records: 3272  Deleted: 0  Skipped: 0  Warnings: 6

mysql> quit;
```

在运行 Sqoop 的 `import` 命令之前，使用 `jps` 命令验证 HDFS 和 YARN 是否已启动：

```
~$ sudo su hadoop
hadoop@ubuntu:~$ jps

4051 NodeManager
31134 Jps
3523 DataNode
3709 SecondaryNameNode
3375 NameNode
3921 ResourceManager
```

此时，使用 `import` 命令将表 `average_price_by_state` 中的数据导入 HDFS。可以分别通过 `--connect` 选项、`--username` 选项和 `--table` 选项来指定源数据库的连接字符串、用户名和表名。将可选的 `-m` 标志设置为 1，表示该作业使用单个 map 任务：

```
/srv/sqoop$ sqoop import --connect jdbc:mysql://localhost:3306/energydata
    --username root --table average_price_by_state -m 1

15/01/20 22:47:35 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6
15/01/20 22:47:35 INFO manager.MySQLManager: Preparing to use a MySQL
    streaming resultset.
15/01/20 22:47:35 INFO tool.CodeGenTool: Beginning code generation
15/01/20 22:47:36 INFO manager.SqlManager: Executing SQL statement:
    SELECT t.* FROM `average_price_by_state` AS t LIMIT 1

(output truncated)

15/01/25 22:47:53 INFO mapreduce.ImportJobBase: Transferred 200.4287 KB in
    15.3718 seconds (13.0387 KB/sec)
15/01/25 22:47:53 INFO mapreduce.ImportJobBase: Retrieved 3272 records.
```

在本例中，因为表不包含主键，而主键是分割和合并多个 map 任务所必需的，所以需要指定 `import` 命令使用单个 map 任务。因为指定了导入任务使用一个 map 任务，所以 HDFS 中只有一个文件：

```
/srv/sqoop$ hadoop fs -head average_price_by_state/part-m-00000 | head
```

```
2012,AK,Total Electric Industry,17.88,14.93,16.82,0.00,null,16.33
2012,AL,Total Electric Industry,11.40,10.63,6.22,0.00,null,9.18
2012,AR,Total Electric Industry,9.30,7.71,5.77,11.23,null,7.62
2012,AZ,Total Electric Industry,11.29,9.53,6.53,0.00,null,9.81
2012,CA,Total Electric Industry,15.34,13.41,10.49,7.17,null,13.53
2012,CO,Total Electric Industry,11.46,9.39,6.95,9.69,null,9.39
2012,CT,Total Electric Industry,17.34,14.65,12.67,9.69,null,15.54
2012,DC,Total Electric Industry,12.28,12.02,5.46,9.01,null,11.85
2012,DE,Total Electric Industry,13.58,10.13,8.36,0.00,null,11.06
2012,FL,Total Electric Industry,11.42,9.66,8.04,8.45,null,10.44
```

我们现在已经成功将数据从 MySQL 导入 HDFS 了！至此，就可以对导入的数据进行后续的 MapReduce 处理，或将数据加载到另一个 Hadoop 数据源中（例如 Hive、HBase 或 HCatalog）。

7.1.2　从MySQL导入Hive

因为关系数据库（在本例中是 MySQL）中的数据已经是结构化的，所以将这些数据导入 Hive 中与源数据库类似的模式也大有用处，尤其当你打算对数据运行关系查询时。Sqoop 提供了两种方法：一种是先将数据导出到 HDFS，再在 Hive shell 中使用 `LOAD DATA HQL` 命令将其加载到 Hive 中；另一种是使用 Sqoop 直接创建表，并将关系数据库数据加载到相应的 Hive 表中。

使用 `import` 命令，Sqoop 可以根据源数据库定义的模式生成 Hive 表，并将源数据库表中的数据加载进来。但由于 Sqoop 实际上仍然使用 MapReduce 来实现数据加载操作，因此在运行导入工具之前，必须删除所有与输出名称相同的已有数据目录：

```
/srv/sqoop$ hadoop fs -rm -r /user/hadoop/average_price_by_state
```

然后运行 Sqoop 的 `import` 命令，将数据库的 JDBC 连接字符串、表名、字段分隔符、行终止符和 `null` 字符串值传递给它：

```
/srv/sqoop$ sqoop import --connect jdbc:mysql://localhost:3306/energydata
    --username root --table average_price_by_state
    --hive-import --fields-terminated-by ','
    --lines-terminated-by '\n' --null-string 'null' -m 1

(output truncated)

15/01/20 00:14:37 INFO hive.HiveImport: Table default.average_price_by_sta
    [numFiles=2, numRows=0, totalSize=205239, rawDataSize=0]
```

```
15/01/20 00:14:37 INFO hive.HiveImport: OK
15/01/20 00:14:37 INFO hive.HiveImport: Time taken: 0.435 seconds
15/01/20 00:14:37 INFO hive.HiveImport: Hive import complete.
15/01/20 00:14:37 INFO hive.HiveImport: Export directory is empty, removin
```

Hive 会将 double 类型的列转换为 float 类型，并去掉所有 NOT NULL 字段约束。除此之外，Hive 表的结构与 MySQL 的 average_price_by_state 表的初始定义一模一样，名字也一样。



如果同一台机器上也运行着 HBase 并且设置了 HBASE_HOME 环境变量，那么你在运行以上命令时可能会遇到以下错误：

```
INFO hive.HiveImport: Exception in thread "main"
    java.lang.NoSuchMethodError:
    org/apache/thrift/EncodingUtils.setBit(BIZ)B
```

这是 HBase 和 Hive 之间的 Thrift 版本冲突导致的。可以将 HBASE_HOME 临时设置为不存在的路径，然后在导入后重新加载 bash 配置文件，就能避免错误：

```
/srv/sqoop$ export HBASE_HOME=/fake/path

(Sqoop Hive commands)

/srv/sqoop$ source ~/.profile
```

在本地模式下，Hive 将在它运行的文件系统中创建一个 metastore_db 目录；在上一个示例中，metastore_db 是在 SQOOP_HOME（/srv/sqoop）下创建的。打开 Hive shell 并验证表 average_price_by_state 是否已被创建：

```
/srv/sqoop$ hive

hive> DESC average_price_by_state;

OK
```

```
year                int
state               string
sector              string
residential         double
commercial          double
industrial          double
transportation      double
other               double
total               double
Time taken: 0.858 seconds, Fetched: 9 row(s)
```

你还可以通过运行 `COUNT` 查询来验证是否已导入了 3272 行；由于该数据集相对较小，因此也可以通过运行 `SELECT * FROM average_price_by_state` 来验证数据。将数据和模式导入 Hive 之后，就可以通过 Hive 命令行接口或其他 Hive 接口对数据进行后续分析了。

7.1.3　从MySQL导入HBase

HBase 旨在为需要实时访问行级数据的大量并发客户端处理大量数据。尽管在大多数小规模和中等规模的数据应用程序中，关系数据库也能很好地满足这一要求，但如果应用程序需要扩展性更强的存储解决方案，我们就得考虑将一些大规模和重负载的数据组件转移到分布式数据库，比如 HBase。

Sqoop 的导入工具能将数据从关系数据库导入 HBase。与 Hive 一样，有两种导入方法：一种是先导入 HDFS，然后使用 HBase CLI 或 API 将数据加载到 HBase 表中；另一种是使用 `--hbase-table` 选项指示 Sqoop 直接将数据导入 HBase 表。

在这个示例中，要转移到 HBase 的数据是一个博客统计数据表，其中每条记录都包含一个主键（由竖线分隔的 IP 地址和年份）和每个月对应的列（每一列表示该年该 IP 在该月的访问数）。你可以在本书 GitHub 仓库的 /data 目录中找到名为 weblogs.csv.zip 的压缩文件。下载并解压，然后将 CSV 文件加载到 MySQL 表中：

```
~$ mysql -u root -p

mysql> CREATE DATABASE logdata;

mysql> GRANT ALL PRIVILEGES ON logdata.* TO '%'@'localhost';

mysql> GRANT ALL PRIVILEGES ON logdata.* TO ''@'localhost';
```

```
mysql> USE logdata;

mysql> CREATE TABLE weblogs (ipyear varchar(255) NOT NULL PRIMARY KEY,
    january int(11) DEFAULT NULL,
    february int(11) DEFAULT NULL,
    march int(11) DEFAULT NULL,
    april int(11) DEFAULT NULL,
    may int(11) DEFAULT NULL,
    june int(11) DEFAULT NULL,
    july int(11) DEFAULT NULL,
    august int(11) DEFAULT NULL,
    september int(11) DEFAULT NULL,
    october int(11) DEFAULT NULL,
    november int(11) DEFAULT NULL,
    december int(11) DEFAULT NULL);

mysql> quit;

~$ mysql -u root -p logdata --local-infile=1

mysql> LOAD DATA LOCAL INFILE '/home/hadoop/hadoop-fundamentals/data/weblo
    INTO TABLE weblogs FIELDS TERMINATED BY ','
    LINES TERMINATED BY '\n' IGNORE 1 LINES;

Query OK, 27300 rows affected (0.20 sec)
Records: 27300  Deleted: 0 Skipped: 0 Warnings: 0

mysql> quit;
```

与之前一样，需要验证 Hadoop 和 HBase 守护进程是否正在运行：

```
~$ cd $HBASE_HOME
/srv/hbase$ bin/start-hbase.sh
```

然后，运行 Sqoop 的 `import` 命令，将数据库的 JDBC 连接字符串、表名、HBase 表名、列族名称和行键名称传递给它：

```
sqoop import --connect jdbc:mysql://localhost:3306/logdata
    --table weblogs --hbase-table weblogs --column-family traffic
    --hbase-row-key ipyear --hbase-create-table -m 1

(output truncated)

15/01/20 00:33:01 INFO mapreduce.ImportJobBase: Transferred 0 bytes in
19.0716 seconds (0 bytes/sec)
15/01/20 00:33:01 INFO mapreduce.ImportJobBase: Retrieved 27300 records.
```

导入的 MapReduce 作业完成后，你应该会看到控制台消息 `INFO mapreduce.ImportJobBase: Retrieved 27300 records`。可以在 HBase shell 中使用 `list` 和 `scan` 命令验证 HBase 表和行是否已被成功导入：

```
/srv/sqoop$ cd $HBASE_HOME
/srv/hbase$ bin/hbase shell

hbase(main):001:0> list

TABLE
linkshare
weblogs
2 row(s) in 1.2900 seconds

=> ["linkshare", "weblogs"]

hbase(main):002:0> scan 'weblogs', {'LIMIT' => 50}
(output truncated)
```

我们已经使用 Sqoop 的导入工具成功将关系数据从 MySQL 导入 HDFS、Hive 和 HBase。实际上，这种工具生成了一个 Java 类，它封装了导入表的行模式。4 该类在导入过程中被 Sqoop 使用，但也可用于后续的 MapReduce 数据处理。因此，除了自动交换 Hadoop 和关系数据库的数据之外，Sqoop 还有助于快速开发与 Hadoop 兼容的其他数据源的处理流水线。若想获取更多有关 Sqoop 特性和功能的信息，建议阅读 Apache Sqoop User Guide（http://sqoop.apache.org/docs/1.4.6/SqoopUserGuide.html）。

4参见 Sqoop User Guide 中的“Basic Usage”（http://bit.ly/1r25bkq）。

7.2 使用Flume获取流式数据

Flume 的设计初衷是从多个数据流中采集和获取大量数据到 Hadoop。Flume 的一个非常常见的用例是采集日志数据，例如采集 Web 服务器上由多个应用服务器发射的日志数据，将其聚合在 HDFS 中，供后续搜索或分析使用。但是，Flume 并不仅限于简单地消费和获取日志数据源，它还可以被自定义为从任何事件源传输大量的事件数据。在这两种情况下，Flume 使我们能够在数据写入 **Hadoop** 后增量且持续地获取流式数据，而不用编写自定义的客户端应用程序将数据批量加载到 HDFS、HBase 或其他 Hadoop 数据槽中。Flume 提供了一种统一而灵活的方法，将数据从大量不同且快速流动的数据流推

送到 Hadoop。

Flume 的灵活性源自其固有的可扩展式数据流架构。除灵活性以外，Flume 旨在通过其分布式架构来保持容错性和可扩展性。尽管一般推荐使用默认的“端到端”可靠性模式（保证所有接收的事件最终都能发送出去）设置 5，但 Flume 还是提供了多种冗余和恢复机制。

> 5参见 Flume User Guide（https://flume.apache.org/FlumeUserGuide.html#reliability）。

我们介绍了 Flume 的总体特征，但为了了解如何构建 Flume 数据流，我们需要查看它的基本构建单元：Flume 代理。

7.2.1 Flume数据流

Flume 将从起点到目的地的数据采集路径表示为数据流。在数据流中，一个数据单元（又称事件，例如一条日志）从源流经一系列跃点（hop）到达下一个目的地（https://flume.apache.org/FlumeUserGuide.html#data-flow-model）。就连 Flume 数据流最简单的实体—— Flume 代理，也体现了数据流这一概念。Flume 代理（实际上是一个 JVM 进程）是 Flume 数据流中的一个单元，一旦外部源发出事件，事件就通过代理传播。代理由三个可配置的组件构成：源、通道和数据槽，如图 7-1 所示。

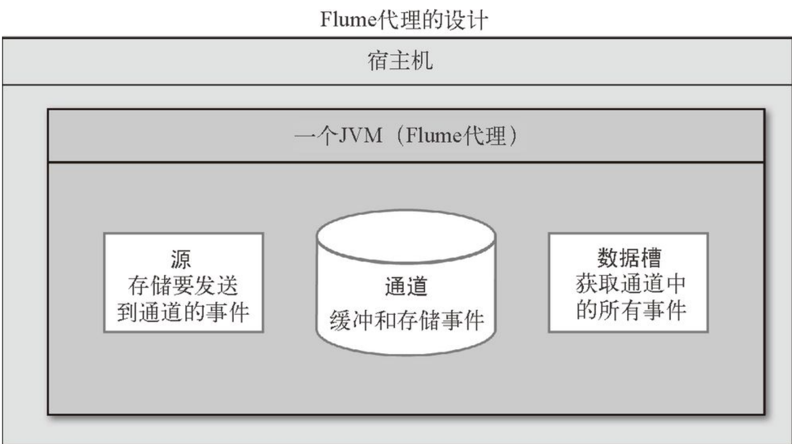


图 **7-1**：**Flume** 代理的设计

Flume 源用于监听并消费来自一个或多个外部数据源（不要与 Flume

源混淆）的事件，通过为每个数据源设置名称、类型和额外的可选参数进行配置。例如，通过运行 `tail -f / etc/httpd/logs/access_log` 命令，可以将 Flume 代理的源配置为接受来自 Apache 访问日志的事件。这种类型的源称为 `exec` 源，因为它需要 Flume 执行 Unix 命令来检索事件。

当代理消费一个事件时，Flume 源将其写入通道。该通道作为存储队列，存储和缓冲事件，直到它们可以被读取。事件以事务性方式写入通道，这意味着直到事件被消费并且相应的事务被明确地关闭，通道都将把所有事件保存在队列中。因此，即便一个代理停止，Flume 也能够保持数据事件的持久性。

Flume 数据槽最终从通道中读取和移除事件，并将其转发到下一个跃点或最终目的地。**6** 因此，可以将数据槽配置为将其输出作为流式数据源写入另一个 Flume 代理或数据存储（例如 HDFS 或 HBase）。Flume 支持多种内置数据槽，在 Apache Flume User Guide（https://flume.apache.org/FlumeUserGuide.html#flume-sinks）中均有记录。

6详见《Flume：构建高可用、可扩展的海量日志采集系统》，Hari Shreedharan 著。

通过使用这种源 - 通道 - 数据槽模式，可以轻松构建一个简单的 Flume 单代理数据流，从 Apache 访问日志中消费事件，并将日志事件写入 HDFS，如图 7-2 所示。

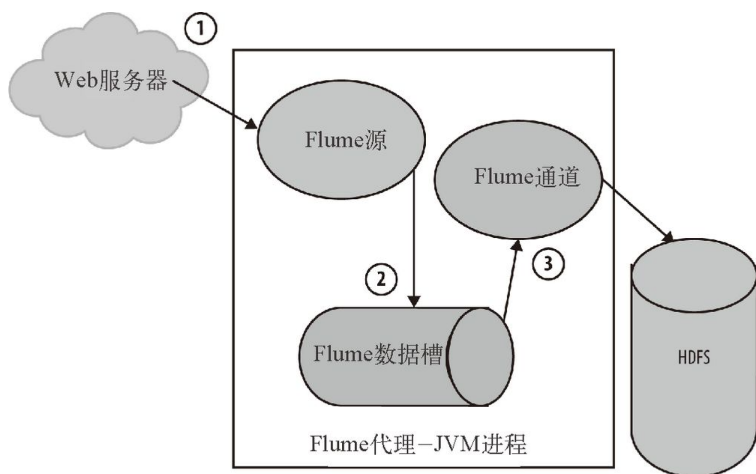


图 **7-2**：简单的 **Flume** 数据流

但是由于 Flume 代理的适配性很强，它甚至可以被配置为多个源、通道和数据槽，因此，可以通过将多个 Flume 代理链接在一起来构建多代理数据流，如图 7-3 所示。

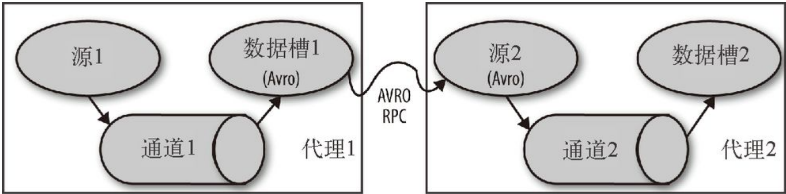


图 7-3：Flume 多代理数据流

尽管在解决流式数据处理架构时，Flume 有一些处理常见场景的数据流拓扑模式，但是将 Flume 代理组织成复杂数据流的方式绝不仅限于此。以日志采集中的一个常见场景为例，生成日志的大量客户端将事件写入多个 Flume 代理，这被称为“第一层”代理，因为它们消费外部数据源层的数据。**7** 如果要将这些事件写入 HDFS，可以将每个第一层代理的数据槽设置为写入 HDFS；但在第一层扩展时，这可能会引起一些问题。由于几个不同的代理分别写入 HDFS，因此该数据流将无法处理周期性暴增的存储系统数据写入，从而可能引发负载和延迟的峰值。

> 7详见《Flume：构建高可用、可扩展的海量日志采集系统》，Hari Shreedharan 著。

通过添加第二层代理，可以汇总和缓冲来自第一层的事件，从而更好地将第一层代理与数据槽（HDFS）隔离。因此，第二层代理一方面可以聚合接收到的事件，降低调试难度；另一方面，它又可以控制对存储系统的写入速率，使得整个数据流可以吸收更长、更大的负载峰值。**8** 这种拓扑模式称为 **fan-in** 流，如图 7-4 所示。

> 8同上。

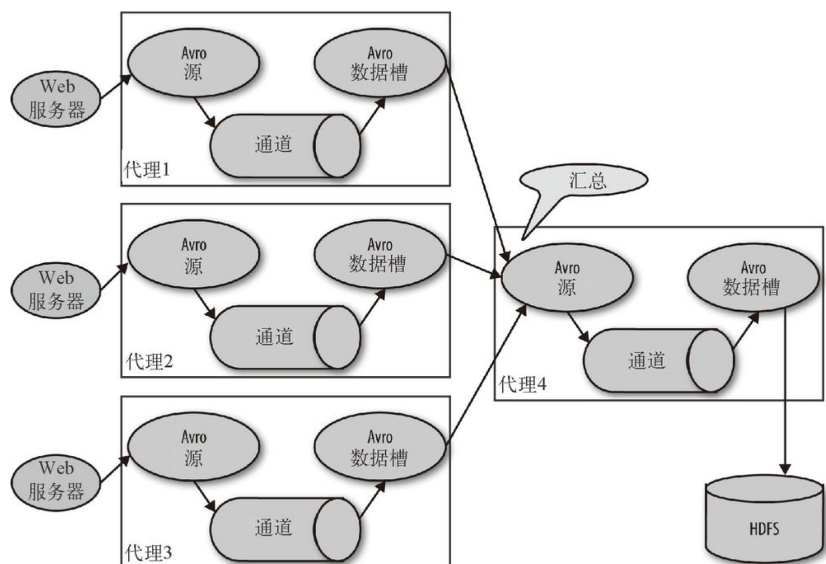


图 **7-4**：**Flume** 的 **fan-in** 数据流

你可能猜到了，随着产生数据的服务器的增加，第一层代理、后续代理以及层的数量通常也会相应地增加。虽然调优和设计这样复杂的 Flume 架构已经超出本书的范围，但如果你有兴趣进一步了解 Flume 的设计原则，推荐你阅读 Hari Shreedharan 的《Flume：构建高可用、可扩展的海量日志采集系统》。在下一小节中，我们将配置一个简单的 Flume 单代理数据流来获取自定义日志。

7.2.2　使用Flume获取产品印象数据

能获取 Apache 访问日志（http://bit.ly/1r278NW）之类的标准日志数据或 Twitter firehose

（http://bit.ly/1r27c04）流式数据的 Flume 单代理数据流比比皆是，但 Flume 其实也非常适用于获取自定义数据流，例如自定义应用程序生成的实时分析数据。

本小节的例子将使用 Flume 来消费某虚构在线商店生成的流式用户交互数据。许多电子商务公司都在找寻测量其在线商店微转化率的方法，追踪在线营销的效果。以下指标可用于衡量在线商店的整体效能。**9**

9Ron Kohavi and Foster Provost,“Applications of Data Mining to Electronic Commerce”,

Data Mining and Knowledge Discovery 5:1–2 (2001): 5–10.

点击率

产品印象转化为点击（或点击链接 / 图片浏览产品详情页面的操作）的次数。

加入购物车率

点击转化为加入购物车的次数。

购物车购买率

加入购物车转化为购买行为的次数。

购买率

产品印象最终转化为购买行为的百分比。

通常，要获取这些指标，就需要捕获细粒度的产品印象，也就是在电子商务 Web 应用程序中插桩，记录访问者与产品的每个交互。这些交互包括查看产品链接、点击进入产品详情页面、向 / 从购物车添加 / 移除产品以及购买产品。在写入数据后，以一定的间隔提取和分析数据，从而生成报表、调整产品特征、驱动个性化体验等。

我们将模拟一个电子商务印象日志，以如下 JSON 格式记录用户与产品的交互信息：

```
{
    "sku": "T9921-5",
    "timestamp": 1453167527737,
    "cid": "51761",
    "action": "add_cart",
    "ip": "226.43.51.25"
}
```

动作类型包括 view、click、add_cart、remove_cart 和 purchase。在本书 GitHub 仓库的 /flume 目录中可以找到一个生成样本印象日志的脚本文件，通过执行以下操作来运行该文件：

```
$ ./impression_tracker.py
```

这将输出并创建一个名为 impressions.log 的文件，并将其放在 /tmp/impressions/ 路径下。使用拥有 sudo 权限的用户运行 setup.sh 脚本，在本地文件系统和 HDFS 中创建必需的目录：

```
$ ./setup.sh
```

本示例模拟了一个简单的 Flume 双代理数据流，我们在其中建立了一个客户端代理，它在 Web 服务器上运行，获取 impressions.log 中的记录并将这些事件发送到一个 Avro 数据槽。Avro 是一种轻量级的 RPC 协议，也提供了简单的数据序列化功能。它使我们能轻松地设置 RPC 协议，将客户端代理的数据槽中的数据发送到采集代理的源。然后，采集代理将这些事件写入 HDFS。最终的流程如图 7-5 所示。

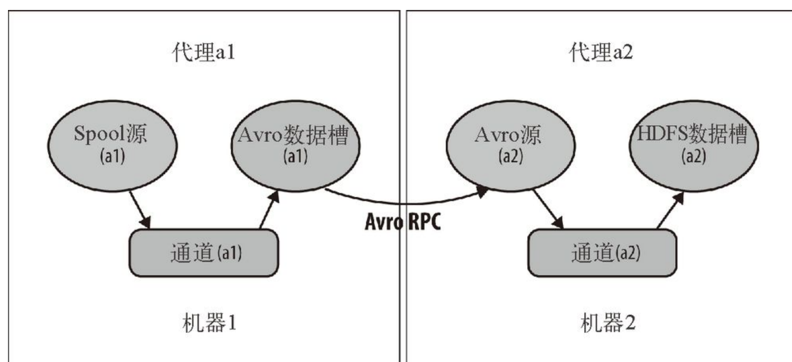


图 **7-5**：采集日志到 **HDFS**

设置 Flume 代理从编写配置文件开始。如前所述，所有 Flume 代理都由源、通道和一个或多个数据槽组成。首先将客户端代理的源配置为印象日志的位置：

```
# 定义spooling目录源
client.sources=r1
client.sources.r1.channels=ch1
client.sources.r1.type=spooldir
client.sources.r1.spoolDir=/tmp/impressions
```

将源名称设置为 `r1`，稍后将使用它来引用和设置源的其他属性。然后，为源指定一个已命名的通道，本例将其命名为 `ch1`。另外，将 `r1` 的源配置为 `spooldir` 类型，用于从磁盘指定的“假脱机”目录中获取数据。这个源将会监视指定目录中的新文件，当新文件出现时，

从中解析出事件。在给定的文件被完全读入通道之后，它会被重命名，表明已被 Flume（http://bit.ly/flume-spool）完全获取。

接着，来配置客户端代理的通道，它会缓冲从源到数据槽的数据。设置一个名为 `ch1` 的通道，并将其类型设置为 `FILE`。默认情况下，文件通道通过将数据写入用户主目录下名为 checkpoint 和 data 的目录下的文件来缓存数据。可以通过配置 `checkpointDir` 和 `dataDirs` 值来覆盖特定通道的这些文件路径：

```
# 定义一个文件通道
client.channels=ch1
client.channels.ch1.type=FILE
```

最后，需要为客户端代理配置数据槽。在这个示例中，客户端代理将其数据写入 Avro 数据槽。我们将其命名为 `k1`，并将其配置为从 `ch1` 通道获取数据。Avro 数据槽需要一个主机名和端口：

```
# 定义一个Avro数据槽
client.sinks=k1
client.sinks.k1.type=avro
client.sinks.k1.hostname=localhost
client.sinks.k1.port=4141
client.sinks.k1.channel=ch1
```

接下来，配置采集代理，它从之前配置的 Avro 源消费事件，并将这些事件写入 HDFS。源、通道和数据槽的配置如下所示：

```
#定义一个Avro源
collector.sources=r1
collector.sources.r1.type=avro
collector.sources.r1.bind=0.0.0.0
collector.sources.r1.port=4141
collector.sources.r1.channels=ch1

#定义一个文件通道，为了可靠性，使用多个磁盘
collector.channels=ch1
collector.channels.ch1.type=FILE
collector.channels.ch1.checkpointDir=/tmp/flume/checkpoint
collector.channels.ch1.dataDirs=/tmp/flume/data

#定义HDFS数据槽，将事件持久化为文本
collector.sinks=k1
collector.sinks.k1.type=hdfs
collector.sinks.k1.channel=ch1
```

请注意，只要类型、绑定主机和端口的配置一致，源名称不需要与客户端代理的数据槽名称相匹配。我们也为此代理配置了一个文件通道，但是覆盖了检查点和数据目录，避免与客户端代理的通道发生冲突；此外，还声明了一个名为 k1、类型为 hdfs 的数据槽，它从为此代理配置的 ch1 通道消费事件。

HDFS 数据槽需要一个 path 配置，用于指定代理写入数据的 HDFS 位置。此外，我们还指定了其他一些可选配置参数，比如预期文件名的前缀和后缀、文件格式，以及每一批次写入的最大事件数：

```
# HDFS数据槽配置
collector.sinks.k1.hdfs.path=/user/hadoop/impressions
collector.sinks.k1.hdfs.filePrefix=impressions
collector.sinks.k1.hdfs.fileSuffix=.log
collector.sinks.k1.hdfs.fileType=DataStream
collector.sinks.k1.hdfs.writeFormat=Text
collector.sinks.k1.hdfs.batchSize=1000
```

客户端和采集代理都已完全配置好，可以运行 Flume 代理来执行完整的数据流了。首先，确保你已运行 setup.sh 脚本，并在 /tmp/impressions 下生成了 impressions.log 文件。然后，在终端打开三个选项卡。在第一个选项卡中，导航到 Flume 配置文件并运行如下命令：

```
$ flume-ng agent -n collector --conf . -f collector.conf
```

采集代理启动，等待从客户端代理接收事件。现在，在第二个选项卡中启动客户端代理：

```
$ flume-ng agent -n client --conf . -f client.conf
```

一旦客户端代理完全处理了 impressions.log 文件，你就会看到一条控制台消息，表明 impressions.log 文件已被完全处理并重命名为 impressions.log.COMPLETED：

```
INFO avro.ReliableSpoolingFileEventReader: Preparing to move file
/tmp/impressions/impressions.log to
/tmp/impressions/impressions.log.COMPLETED
```

然后检查第一个选项卡，验证采集代理是否已处理所有事件并将其写入了 HDFS。确认日志已写入配置指定的 HDFS 源目录：

```
$ hadoop fs -ls /user/hadoop/impressions/
$ hadoop fs -cat /user/hadoop/impressions/impressions.1453085307781.log |
```

文件前缀为 impressions，后缀为 .log 扩展名，但中间的时间戳会随你运行 Flume 数据流的日期和时间而变。虽然这个双代理数据流演示的 Flume 多代理数据流非常简单，但是 Flume 为许多其他类型和配置的源、通道和数据槽提供了丰富的支持，以实现更复杂和可扩展的数据流。关于这一点，可以查阅 Flume 的用户文档（https://flume.apache.org/FlumeUserGuide.html）。

7.3　小结

在本章中，我们学习了如何使用 Sqoop 将批量数据从关系数据库高效地传输到各种 Hadoop 数据存储中。有关集成 Sqoop 的更多信息，推荐你阅读 Kathleen Ting 和 Jarek Jarcec Cecho 合著的 *Apache Sqoop Cookbook*（http://shop.oreilly.com/product/0636920029519.do）。我们还了解了 Flume 如何以可靠、可扩展的方式将流式数据传输到 Hadoop。如果你想了解更多关于 Flume 流的配置和架构的信息，推荐你阅读 Hari Shreedharan 的《Flume：构建高可用、可扩展的海量日志采集系统》。

虽然 Sqoop 和 Flume 位列 Hadoop 最常用的数据采集工具之中，但是在数据采集和流处理领域还有许多本章未涉及的 Hadoop 生态系统项目。Apache Kafka 就是其中之一，它虽然不是专门为 Hadoop 设计的，但却支持将数据高吞吐、并行地加载到 Hadoop 中。除了 Flume 和 Kafka 之外，开发采集和处理 Hadoop 和 Spark 中实时流式数据的工具和模式也是目前的研究重点。如果想进一步了解使用这些工具进行数据采集的实际用例，推荐你阅读《Hadoop 应用架构》**10** 中的“Hadoop 数据移动”一章。

> 10本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1710。——编者注

第 8 章　使用高级 API 进行分析

第 6 章解释了为什么要放弃原生 MapReduce，转而使用 Hive 这样的较高级语言的部分原因——因为前者实现起相对简单的操作也可能十分困难、笨拙和冗长。即便是经验丰富的 Java 和 MapReduce 程序员也会发现，大多数严谨的 Hadoop 应用程序的开发周期都很长，需要编写多个 mapper 和 reducer 并将它们链接起来，形成复杂的作业链或数据处理工作流。

此外，由于 MapReduce 旨在以批处理方式运行，因此它在运行需要响应反馈的迭代处理（包括许多机器学习算法）和交互式数据挖掘的数据分析时，会有许多限制。原生 MapReduce 在开发效率、维护和运行时表现出的性能方面的不足引发了对 Hadoop 更高层次的抽象，甚至是扩展 MapReduce 范式的新处理引擎的需求。

本章将介绍 Pig，它是 MapReduce 的一种编程抽象，有助于构建基于 MapReduce 的数据流；此外，还将介绍一些扩展核心 RDD API 的新 Spark API，让开发人员能使用他们熟悉的基于 SQL 的概念和语法，降低计算结构化数据的难度。这些项目将提供表达力强大的 API，使分析人员仅凭几行代码就能构建复杂的应用程序，从而提高 MapReduce 和 Spark 应用程序的开发效率。

8.1 Pig

和 Hive 一样，Pig 也是一种 MapReduce 抽象。它允许用户用更高级的语言去表达数据处理和分析操作，然后这些操作被编译成一个 MapReduce 作业。Pig 是雅虎开发的一个工具，通过将脚本表示为数据流 1，使研究人员和工程师能更轻松地编写数据挖掘 Hadoop 脚本。Pig（http://pig.apache.org）现在是一个 Apache 顶级项目，包含两个主要平台组件。

> 1《Hadoop 权威指南（第 4 版）》，Tom White 著（O'Reilly）。

- Pig Latin，用于表达数据流的过程式脚本语言。
- 运行 Pig Latin 程序的 Pig 执行环境，可以在本地或 MapReduce 模式下运行，包含 Grunt 命令行接口。

Hive 的 HQL 充分继承了 SQL 的声明式风格，但 Pig Latin 不同，它本质上是过程式的，旨在让程序员能轻松实现一系列应用于数据集的数据操作和转换，从而形成数据流水线。2 虽然 Hive 非常适用于能很好转换为基于 SQL 的脚本的用例，但要转换多重复杂数据时，SQL 就显得力不从心了。Pig Latin 是实现这类多级数据流的理想选择，特

别是需要从多个源聚合数据，并在数据处理流程的每个阶段执行后续转换的情况下。

> 2Alan Gates,“Comparing Pig Latin and SQL for Constructing Data Processing Pipelines”(http://yhoo.it/1r2bK6I), Yahoo Developer Network's Hadoop Blog, January 29, 2010.

Pig Latin 脚本从数据开始，对数据应用转换，最后描述所需结果，并将整个数据处理流程作为已优化的 MapReduce 作业执行。此外，Pig 支持使用由 Java、Python、JavaScript 及其他支持的语言编写的用户定义函数（UDF）集成自定义代码。3 因此，我们可以使用相对简单的构造对大数据进行几乎所有的转换和临时分析。

> 3参见 Apache Pig 的文档（http://pig.apache.org/docs/r0.12.0/start.html）。

一定要记住，Pig 和 Hive 一样，最终将编译成 MapReduce 作业，无法突破 Hadoop 批处理方法的局限性。不过 Pig 也确实为我们提供了强大的工具，可以轻松、简便地编写复杂的数据处理流程，以及在 Hadoop 上构建真实业务应用程序所需的细粒度控制。下一节将先回顾一些 Pig 的基本组件，然后通过实现原生的 Pig Latin 运算符和定制函数，对 Twitter 数据执行一些简单的情绪分析。假设你已经在伪分布式模式的 Hadoop 上安装了 Pig。安装 Pig 的步骤可以在附录 B 中找到。

8.1.1　Pig Latin

既然已经设置好了 Pig 和 Grunt shell，那就先来研究一个 Pig 脚本，探讨探讨 Pig Latin 提供的命令和表达式。以下脚本将加载一个星期内带有标签 #unitedairlines 的 Twitter 推文。

> 你可以在 GitHub 仓库的 data/sentiment_analysis/ 文件夹下找到该脚本和相应的数据。

数据文件 united_airlines_tweets.tsv 提供了 tweet ID、固定链接、发布日期、推文和 Twitter 用户名。该脚本加载字典文件 dictionary.tsv，该文件包含已知的“正面的”（positive）和“负面的”（negative）词，以及与每个词相关联的情绪评分（分别为 1 和 -1）。然后，该脚本执行一系列 Pig 变换，生成每个推文的情绪评分和分类（POSITIVE 或 NEGATIVE）：

```
grunt> tweets = LOAD 'united_airlines_tweets.tsv' USING PigStorage('\t')
    AS (id_str:chararray, tweet_url:chararray, created_at:chararray,
    text:chararray, lang:chararray, retweet_count:int, favorite_count:int,
    screen_name:chararray);
grunt> dictionary = LOAD 'dictionary.tsv' USING PigStorage('\t')
    AS (word:chararray, score:int);
grunt> english_tweets = FILTER tweets BY lang == 'en';
grunt> tokenized = FOREACH english_tweets GENERATE id_str,
    FLATTEN( TOKENIZE(text) ) AS word;
grunt> clean_tokens = FOREACH tokenized GENERATE id_str,
LOWER(REGEX_EXTRACT(word, '[#@]{0,1}(.*)', 1)) AS word;
grunt> token_sentiment = JOIN clean_tokens BY word, dictionary BY word;
grunt> sentiment_group = GROUP token_sentiment BY id_str;
grunt> sentiment_score = FOREACH sentiment_group
    GENERATE group AS id, SUM(token_sentiment.score) AS final;
grunt> classified = FOREACH sentiment_score
    GENERATE id, ( (final >= 0)? 'POSITIVE' : 'NEGATIVE' ) AS classificati
    final AS score;
grunt> final = ORDER classified BY score DESC;
grunt> STORE final INTO 'sentiment_analysis';
```

让我们将脚本分解成数据处理流程的单个步骤。

01. 关系和元组

脚本的前两行将数据从文件系统加载到关系 `tweets` 和 `dictionary` 中：

```
tweets = LOAD 'united_airlines_tweets.tsv' USING PigStorage('\t')
    AS (id_str:chararray, tweet_url:chararray, created_at:chararray,
    text:chararray, lang:chararray, retweet_count:int, favorite_coun
    screen_name:chararray);

dictionary = LOAD 'dictionary.tsv' USING PigStorage('\t') AS (word:c
score:int);
```

Pig 中的关系在概念上类似于关系数据库中的表，但它不是有序集合或行，而是一系列无序的元组。元组是一个有序的字段集合。一定要注意，虽然关系声明在赋值的左侧，与典型编程语言中的变量很像，但关系不是变量。给关系别名是为了引用，但它们其实代表的是数据处理流程中的检查点数据集。

首先，使用 `LOAD` 运算符指定文件的文件名（在本地文件系统或 HDFS 上），将它加载到 `tweets` 和 `dictionary` 关系中；然后，使用 `USING` 子句与 `PigStorage` 加载函数指定

文件由制表符分隔；最后，使用 AS 子句定义每个关系的模式，并为每个字段指定列别名和相应的数据类型，但这步不是必需的。即使没有定义模式，仍然可以通过使用 Pig 的位置列（第一个字段为 $0，第二个字段为 $1，依此类推）引用关系中每个元组的字段。如果加载的数据有许多列，但你仅想引用其中几列时，不定义模式可能效果更好。

02. 过滤

下一行对 tweets 关系执行了简单的 FILTER 数据转换，以过滤掉所有不是英文的元组：

```
english_tweets = FILTER tweets BY lang == 'en';
```

FILTER 运算符根据某种条件从关系中选择元组，通常用它来选择所需的数据，或者过滤掉（删除）不想要的数据。因为“lang”字段类型为 chararray，而 chararray 等价于 Java 中的 String 数据类型，所以使用 == 比较运算符来筛选出值为 **en**（English）的记录。结果存储在一个名为 english_tweets 的新关系中。

03. 投影

过滤数据而只保留英文推文（毕竟我们的字典是英文的）之后，我们需要将推文分成单词令牌，这样就可以将单词令牌与字典进行匹配，并可对单词进行一些额外的数据清理，删除 # 和 @：

```
tokenized = FOREACH english_tweets GENERATE id_str,
FLATTEN( TOKENIZE(text) ) AS word;
clean_tokens = FOREACH tokenized GENERATE id_str,
LOWER(REGEX_EXTRACT(word, '[#@]{0,1}(.*)', 1)) AS word;
```

Pig 提供了 FOREACH ... GENERATE 操作来处理关系或集合中的数据列，并将一组表达式应用于集合中的每个元组。GENERATE 子句包含值和／或评估表达式，评估表达式将导出一个新的元组集合并将其传递到流水线的下一步。在此示例中，我们从 english_tweets 关系中投影 id_str 键，并使用 TOKENIZE 函数将 text 字段分割成单词令牌（使用空格分割）。FLATTEN 函数将生成的元组集合中的每个单词提取

出来，id_str 与其中每一个单词构成一个元组。

我们生成的元组集合实际上是 Pig 中的一种特殊数据类型，被称为 **bag**。它代表一个无序的元组集合，类似于关系。但关系被称为“outer bag”，因为关系不能嵌套在另一个 bag 中。在 FOREACH 命令中，结果产生一个叫作 tokenized 的新关系，它的第一个字段是 stock_tweet ID（id_str），第二个字段是由单个单词元组组成的 bag。

然后，对 tokenized 关系再进行投影，取得 id_str 和删除了开头的 # 和 @ 的小写的 word。我们对数据进行了很多转换，所以现在正是验证关系结构是否良好的好时机。你可以随时使用 ILLUSTRATE 运算符来查看基于简明样本数据集生成的每个关系的模式（为减少输出而截断了输出）：

```
grunt> ILLUSTRATE clean_tokens;
-------------------------------------------------------------------------
| tweets | id_str:chararray | tweet_url:chararray                  |
-------------------------------------------------------------------------
|        | 474415416874250240 | https://.../474415416874250240 |
-------------------------------------------------------------------------
```

在设计 Pig 流时，定期使用 ILLUSTRATE 命令有助于了解查询的状态，并验证流水线中的每个检查点。

04. 分组和连接

我们已经对所选的推文进行了分词，也清洗了单词令牌，现在希望 JOIN 得到的令牌和字典，根据单词令牌进行匹配：

```
token_sentiment = JOIN clean_tokens BY word, dictionary BY word;
```

Pig 提供 JOIN 命令，基于公共字段值对两个或多个关系执行连接。虽然默认情况下使用内连接，但是内连接和外连接都是被支持的。示例基于 word 字段对 clean_tokens 关系和 dictionary 关系执行内连接，这将生成一个名为 token_sentiment 的新关系，它包含两个关系的字段：

```
--------------------------------------------------------------------------
| token_sentiment | clean_tokens::id_str:chararray |
```

```
clean_tokens::word:chararray |
dictionary::word:chararray | dictionary::score:int |
---------------------------------------------------------------
| | 473233757961723904 | delayedflight | delayedflight | -1 |
---------------------------------------------------------------
```

现在，通过 Tweet ID（即 `id_str`）`GROUP` 这些行，这样稍后才能计算每条推文的总分：

```
sentiment_group = GROUP token_sentiment BY id_str;
```

`GROUP` 运算符将组键（`id_str`）相同的元组分到一起。这个操作会产生一个关系，每个组包含一个元组，每个元组包含两个字段。

- 第一个字段名为“group”（不要将它与 `GROUP` 运算符弄混了），与组键的类型相同。
- 第二个字段采用原先关系的名称（`token_sentiment`），类型为 bag。

现在可以进行数据的最终聚合，计算按 ID 分组的每条推文的总分了：

```
sentiment_score = FOREACH sentiment_group GENERATE group AS id,
 SUM(token_sentiment.score) AS final;
```

然后根据分数，将每条推文分类为“正面”或“负面”：

```
classified = FOREACH sentiment_score GENERATE id,
( (final >= 0)? 'POSITIVE' : 'NEGATIVE' )
    AS classification, final AS score;
```

最后，将结果按分数降序排列：

```
final = ORDER classified BY score DESC;
```

至此，我们定义了情绪分析所需的所有操作和预测。下一节将把这些数据保存在 HDFS 上的一个文件中，可供之后查看、分析结果使用。

07. 存储和输出数据

我们已经对数据应用了所有必要的转换，现在想在某处写出结果。为了实现这一操作，Pig 提供了 STORE 语句。它能获取某个关系，并将结果写入指定位置。默认情况下，STORE 命令将使用 PigStorage 将数据写入 HDFS 上制表符分隔的文件中。在此示例中，我们将 final 关系的结果转储到 Hadoop 用户目录（/user/hadoop/）中名为 sentiment_analysis 的文件夹中：

```
STORE final INTO 'sentiment_analysis';
```

该目录将包括一个或多个 part 文件：

```
$ hadoop fs -ls sentiment_analysis
Found 2 items
-rw-r--r-- 1 hadoop supergroup          0 2015-02-19 00:10
sentiment_analysis/_SUCCESS
-rw-r--r-- 1 hadoop supergroup       7492 2015-02-19 00:10
sentiment_analysis/part-r-00000
```

在本地模式下，只能创建一个 part 文件；但是在 MapReduce 模式下，part 文件的数量将取决于存储前最后一个作业的并行性。Pig 提供了几个功能，能设置生成的 MapReduce 作业的 reducer 数量；你可以在 Apache Pig 文档（http://pig.apache.org/）中阅读更多有关 Pig 并行功能的内容。

当使用较小的数据集时，使用 grunt shell 将结果快速输出到屏幕，比将结果存储起来更方便。DUMP 命令使用关系的名称将其内容打印到控制台：

```
grunt> DUMP sentiment_analysis;
```

DUMP 命令适用于快速测试和验证 Pig 脚本的输出。但面对大型数据集的输出时，你通常会将结果存储到文件系统供后续分析使用。

8.1.2 数据类型

我们已经介绍了一些 Pig 中的可用嵌套数据结构，比如字段、元组和 bag。Pig 还提供了 **map** 结构，其中包含键值对集合。键始终是 `chararray` 类型，但是值的数据类型不定。在定义推文数据的模式时，我们看到了 Pig 支持的一些原生标量类型。

表 8-1 展示了所有 Pig 支持的标量类型。

表8-1：Pig标量类型

32 位带符号整数
64 位带符号整数
32 位浮点数
64 位浮点数
或字符数组
或字节数组

8.1.3　关系运算符

Pig 通过 Pig Latin 中的关系运算符提供数据操作命令。在之前的示例中，我们使用过其中几个来加载、过滤、分组、投影和存储数据。表 8-2 展示了 Pig 支持的关系运算符。

表8-2：Pig关系运算符

从文件系统或其他存储源加载数据
将关系存入文件系统或其他存储系统
将关系打印到控制台
基于某种条件从关系中选择元组
移除关系中重复的元组
基于数据列生成数据转换
在 Pig 脚本内执行本地 MapReduce 作业
将数据发送给外部脚本或程序
选择一个指定大小的随机样本数据
连接两个或多个关系
将两个或多个关系的数据进行分组
将一个关系的数据进行分组
创建两个或多个关系的向量积
根据一个或多个字段对关系进行排序
限制关系返回的元组数量
合并两个或多个关系
把一个关系切分成两个或多个关系

在 Pig 的用户文档（http://pig.apache.org/docs/r0.14.0/）中可以找到 Pig 的关系运算符、算术、布尔和比较运算符的完整使用语法。

8.1.4 用户定义函数

Pig 最强大的功能之一在于，它能够让用户将 Pig 的原生关系运算符与自己的自定义处理相结合。Pig 为用户定义函数（user-defined function，UDF，http://pig.apache.org/docs/r0.14.0/udf.html）提供了广泛的支持，目前为 6 种语言提供了集成库，分别是 Java、Jython、Python、JavaScript、Ruby 和 Groovy。然而，Java 仍然是最广泛支持的编写 Pig UDF 的语言，并且通常更高效——因为它与 Pig 是相同的语言，可以与 Pig 接口（例如 Algebraic 接口和 Accumulator 接口）集成。

来演示一个之前写过的脚本的简单 UDF。在这种场景下，我们想编写一个自定义的评估 UDF，将分数分类评估转换为一个函数。这样的话，就不需要写：

```
classified = FOREACH sentiment_score GENERATE id,
    ( (final >= 0)? 'POSITIVE' : 'NEGATIVE' )
    AS classification, final AS score;
```

而是写：

```
classified = FOREACH sentiment_score GENERATE id,
    classify(final) AS classification, final AS score;
```

在 Java 中，我们需要扩展 Pig 的 EvalFunc 类并实现 exec() 方法。该方法需要一个元组，并返回一个 String：

```
package com.statistics.pig;

import java.io.IOException;

import org.apache.pig.EvalFunc;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.Tuple;

public class Classify extends EvalFunc {

    @ Override

    public String exec(Tuple args)throws IOException {
```

```
        if (args == null || args.size() == 0) {
            return false;
        }
        try {
            Object object = args.get(0);
            if (object == null) {
                return false;
            }
            int i = (Integer)object;
            if (i >= 0) {
                return new String("POSITIVE");
            } else {
                return new String("NEGATIVE");
            }
        } catch (ExecException e) {
            throw new IOException(e);
        }
    }
}
```

要使用此函数，就需要先编译它，将它打包成 JAR 文件，然后使用 REGISTER 运算符将它注册到 Pig：

```
grunt> REGISTER statistics-pig.jar;
```

然后通过命令调用该函数：

```
grunt> classified = FOREACH sentiment_score GENERATE id,
    com.statistics.pig.Classify(final) AS classification, final AS score;
```

建议你阅读 UDF 的相关文档，其中包含支持的 UDF 接口列表，并提供用于评估、加载、存储、汇总 / 过滤数据任务的示例脚本。Pig 还提供了一组由用户贡献的 UDF，叫作 Piggybank。它们与 Pig 一起发布，但必须注册才能使用。有关 Piggybank 的详细信息，请参见 Apache 文档（https://cwiki.apache.org/confluence/display/PIG/PiggyBank）。

8.1.5 Pig小结

对偏爱过程式编程模型的用户来说，Pig 是一个强大的工具。它能控制流水线中的数据检查点，更提供了对每个步骤如何处理数据的细粒度控制。当你需要更灵活地控制数据流中的操作顺序（例如提取、转换、加载或 ETL 过程），或者面对不适合使用 Hive 的 SQL 语法的半

结构化数据时，Pig 都是一个很好的选择。

8.2　Spark高级API

现在的许多项目和工具都围绕着 MapReduce 和 Hadoop 构建，以支持常见的数据任务并提供更高效的开发人员体验。例如我们已经见过的，使用 Hadoop Streaming 这样的框架以非 Java 语言（如 Python）编写和提交 MapReduce 作业。我们还介绍了为 MapReduce 提供更高级别抽象的工具——Hive 和 Pig。Hive 提供了关系型接口和声明式的基于 SQL 的语言来查询结构化数据，而 Pig 提供了一个在 Hadoop 中编写面向数据流程序的过程式接口。

但在实际工作中，典型的分析工作流将结合关系查询、过程式编程和自定义处理三者。这意味着大多数端到端 Hadoop 工作流都要集成多个不同组件，并在不同编程 API 之间切换。与以 MapReduce 为中心的 Hadoop 栈相比，Spark 在编程上有两大优势。

- 内置表达力强大的 API，以标准的通用语言（例如 Scala、Java、Python 和 R）提供。
- 包含若干内置高级库的统一编程接口，支持广泛的数据处理任务，比如复杂的交互式分析、结构化查询、流处理和机器学习。

第 4 章使用 Spark 的基于 Python 的 RDD API 编写了一个程序，在不使用工具类的大约 10 行代码中，对数据集进行了加载、清洗、连接、过滤和排序。如你所见，Spark 的 RDD API 提供了更丰富的功能操作，代码量远远小于在 MapReduce 中编写的类似程序。然而，因为 RDD 是一种通用的、与类型无关的数据抽象，而且 RDD 固定的模式只有你知道，所以处理结构化数据的过程非常繁琐；这通常将迫使你必须编写大量重复代码才能访问内部数据类型，以及将简单查询操作转换为 RDD 操作的函数语义。考虑图 8-1 所示的操作，该操作试图计算各学科教授的平均年龄。

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	42	B Jones
Chem	61	M Kennedy

RDD API

```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
  .reduceBy Key(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

图 **8-1**：使用 **Spark** 的 **RDD API** 进行聚合

在实践中，使用关系型数据的通用语言 SQL 来处理这样的结构化表格数据要更自然一些。好在 Spark 提供了一个集成的模块，让我们仅用一行简单的代码就表达了前面的聚合，如图 8-2 所示。

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	42	B Jones
Chem	61	M Kennedy

DataFrame API

```
data.groupBy("dept").avg("age")
```

图 **8-2**：使用 **Spark** 的 **DataFrame API** 进行聚合

8.2.1 Spark SQL

Spark SQL 是 Apache Spark 中的一个模块，它提供了一个关系型接

口，让你在 Spark 中使用熟悉的基于 SQL 的操作来处理结构化数据。可以通过 JDBC/ODBC 连接器、内置的交互式 Hive 控制台或其内置的 API 访问 Spark SQL。最后一种访问方式是其中最有趣，也是最强大的，因为实际上，Spark SQL 是作为库在 Spark 的 Core 引擎和 API 之上运行的。所以，可以使用与 Spark 的 RDD API 相同的编程接口访问 Spark SQL 的 API，如图 8-3 所示。

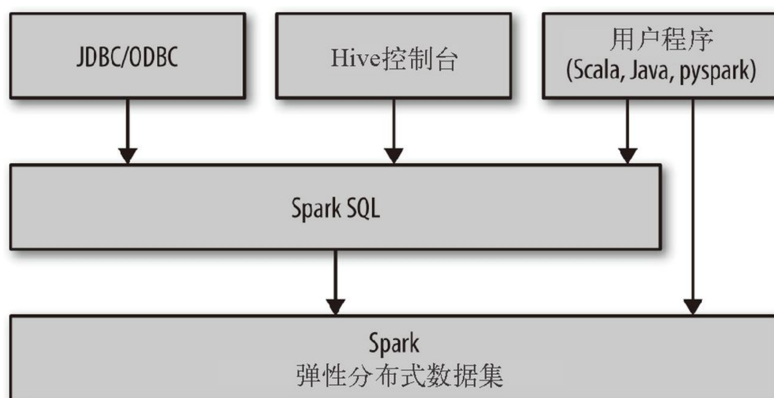


图 **8-3**：**Spark SQL** 接口

这让我们能在一个编程环境中，将关系查询的优势、Spark 过程式处理的灵活性和 Python 分析库的强大功能充分结合并付诸实践。4

> 4Michael Armbrust et al.,"Spark SQL: Relational Data Processing in Spark", ACM SIGMOD Conference 2015.

来写一个简单的程序，用 Spark SQL API 加载 JSON 数据并进行查询。你可以在运行着的 `pyspark` shell 中直接输入这些命令，也可以在使用 `pyspark` 内核的 Jupyter notebook 中输入这些命令；无论使用哪种方法，都要确保有一个运行着的 SparkContext，因为假定变量 `sc` 将引用它。

> 以下示例使用 /sparksql 目录下运行的 Jupyter notebook。确保你已经解压了 GitHub 仓库的 /data 目录中的 sf_parking.zip 文件。可以在 GitHub 仓库的 /sparksql 目录中查看 sf_parking.ipynb 文件。

首先，从 pyspark.sql 包导入 `SQLContext` 类。`SQLContext` 类是 Spark SQL API 的入口，通过包装活动的 `SparkContext` 对象创

建：

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

在此示例中，我们将从 SF Open Data（https://data.sfgov.org）加载一个 JSON 格式的数据集，该数据集列出了 2011 年 9 月旧金山公开可用的路边停车位。5

5SF Open Data,“Off-Street Parking Lots and Parking Garages”(http://bit.ly/1r2n9Do).



与 Hadoop 一样，Spark SQL 也要求将 JSON 数据格式化。所以，要删除第一个和最后一个大括号或中括号，让每个 JSON 对象都包含在一行中，后跟换行符（即没有跨越多行的 JSON 对象）。我们使用实用程序 clean_json.py 为你提供了一个已清洗的数据文件，名为 sf_parking_clean.json。

但对于极大的数据集，你可以使用 Spark 来执行格式化。例如，如果需要手动删除 JSON 文件的第一个和最后一个方括号，可以这样加载和格式化文件：

```
input = sc.wholeTextFiles(input_path).map \
(lambda (x,y): y)
data = input.flatMap(lambda x: json.loads(x))
data.map(lambda x: json.dumps(x)) \
.saveAsTextFile(output_path)
```

`wholeTextFiles` 函数创建了一个 `PairRDD`，其中键是拥有完整路径的文件名（例如“hdfs://localhost:9000/user/hadoop/sf_parking/sf_parking.json”），值是整个文件内容的字符串。使用 `map` 操作提取内容作为输入，然后使用 `flatMap` 将字符串内容读取为 JSON 格式。

调整文件至正确格式后，通过调用 `sqlContext.read.json` 并将文件的路径传递给它，就可以轻松加载文件内容了：



```
parking = sqlContext.read.json('../data/sf_parking/sf_parking_clean.json')
```

也可以将一个目录的路径传递给 sqlContext，sqlContext 会将其中所有的文件加载到 parking 对象中。Spark SQL 自动推断 JSON 数据集的模式，可以使用 printSchema 方法以漂亮的树形式来显示它。

```
parking.printSchema()

root
 |-- address: string (nullable = true)
 |-- garorlot: string (nullable = true)
 |-- landusetyp: string (nullable = true)
 |-- location_1: struct (nullable = true)
 |    |-- latitude: string (nullable = true)
 |    |-- longitude: string (nullable = true)
 |    |-- needs_recoding: boolean (nullable = true)
 |-- mccap: string (nullable = true)
 |-- owner: string (nullable = true)
 |-- primetype: string (nullable = true)
 |-- regcap: string (nullable = true)
 |-- secondtype: string (nullable = true)
 |-- valetcap: string (nullable = true)
```

还可以查看第一行数据：

```
parking.first()

Row(address=u'2110 Market St', garorlot=u'L', landusetyp=u'restaurant',
location_1=Row(latitude=u'37.767378', longitude=u'-122.429344',
needs_recoding=False), mccap=u'0', owner=u'Private', primetype=u'PPA',
regcap=u'13', secondtype=u' ', valetcap=u'0')
```

为了对数据集运行 SQL 语句，必须先将其注册为临时命名表：

```
parking.registerTempTable("parking")
```

这才允许我们运行额外的表和 SQL 方法，比如用表格形式显示前 20 行数据的 show：

```
parking.show()

...output truncated...
```

如果要在 parking 表上执行 SQL 语句的话，就需要使用 sql 方法，并将完整的查询语句传递给它。来运行一个聚合，按照主要类型和次要类型对停车场地进行分组，获得停车场地的数量以及普通停车位的平均个数。将其存储在 aggr_by_type 中，并调用 show() 来查看完整的结果：

```
aggr_by_type = sqlContext.sql("SELECT primetype, secondtype,
                               count(1) AS count,
                               round(avg(regcap), 0) AS avg_spaces " +
                              "FROM parking " +
                              "GROUP BY primetype, secondtype " +
                              "HAVING trim(primetype) != '' " +
                              "ORDER BY count DESC")

aggr_by_type.show()
```

除了 JSON 之外，Spark SQL 还支持其他几种数据源，比如本地文件系统、HDFS 或 S3 中的文件（例如文本文件、parquet 文件、CSV 文件等；CSV 文件可以使用 Databricks 的 CSV-reader 实用程序解析，https://github.com/databricks/spark-csv）、JDBC 数据源（例如 MySQL）和 Hive。此外，Spark 甚至可以作为 Hive 的底层执行引擎使用，只需在活动的 Hive 会话中设置 hive.execution.engine=spark 即可。

但 Spark SQL 模块可不仅仅是一个 SQL 接口而已，它的强大归根到底来源于其底层的数据抽象——DataFrame。

8.2.2 DataFrame

DataFrame 是 Spark SQL 中的底层数据抽象。Python Pandas（http://pandas.pydata.org）和 R（https://www.r-project.org）的用户应该非常熟悉数据框的概念；事实上，Spark 的 DataFrame 与原生的 Pandas（使用 pyspark）和 R 数据框（使用 SparkR，https://spark.apache.org/docs/1.6.0/sparkr.html）是可以互操作的。DataFrame 在 Spark 中也表示已定义模式的数据的表。Spark 的 DataFrame 和 Pandas、R 的数据框的关键区别是，前者实际上是一个包装了 RDD 的分布式集合；你可以将其视为行对象的 RDD。

此外，DataFrame 操作在底层进行了许多优化，不仅将查询计划编译

为可执行代码，而且与硬编码的 RDD 操作相比，性能有显著提升，内存占用的空间也大幅减少。事实上，如果在一个基准测试中，将聚合了 1000 万整数对的 DataFrame 代码和与之等效的 RDD 代码进行运行时间上的对比，你会发现 DataFrame 不仅速度快 4~5 倍，而且还消除了 Python 和 JVM 实现（http://bit.ly/1r2vMhm）的性能差距，如图 8-4 所示。

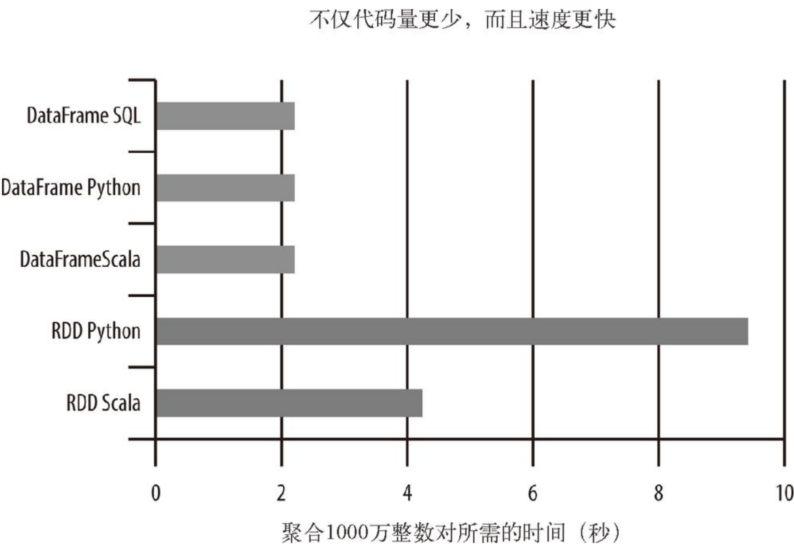


图 **8-4**：**DataFrame** 优化

DataFrame API 简洁直观的语义，加上它计算引擎提供的性能优化，促使 DataFrame 成为了 Spark 所有模块（包括 Spark SQL、RDD、MLlib 和 GraphX）的主要接口。通过这种方式，DataFrame API 提供了统一的引擎，跨越了 Spark 的所有数据源、工作负载和环境，如图 8-5 所示。

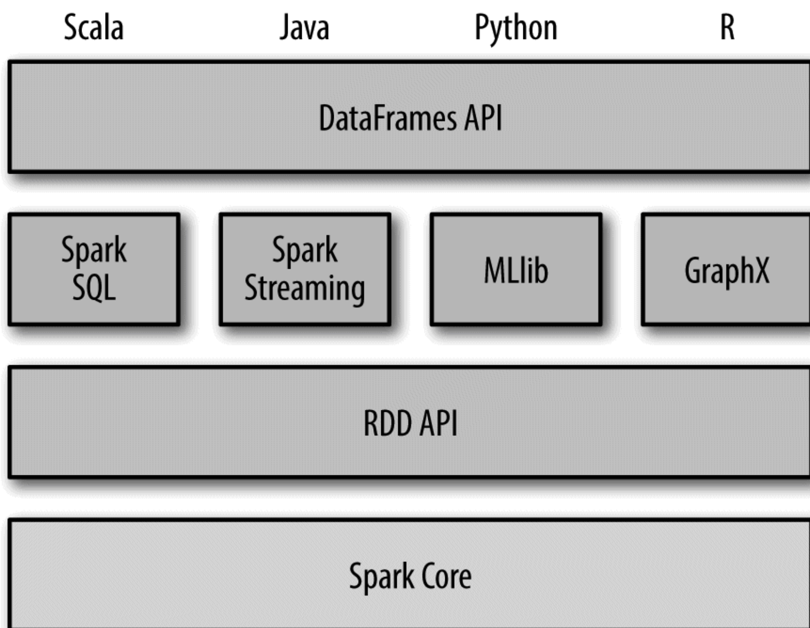


图 8-5：作为 **Spark** 统一接口的 **DataFrame**

在上一个例子中，我们使用 Spark SQL 的 `read` 接口加载了 SF 停车场地数据。但实际上，我们创建了一个叫作 `parking` 的 DataFrame。在那个例子中，我们将 DataFrame 注册为临时表来执行原始 SQL 查询，在 `parking` DataFrame 上也有很多可以调用的关系运算符和窗口函数。事实上，通过将几个简单的 DataFrame 操作连接起来，就可以重写上一个例子中的 SQL 查询：

```
from pyspark.sql import functions as F

aggr_by_type = parking.select("primetype", "secondtype", "regcap") \
                      .where("trim(primetype) != ''") \
                      .groupBy("primetype", "secondtype") \
                      .agg(
                         F.count("*").alias("count"),
                         F.round(F.avg("regcap"), 0).alias("avg_spaces")
                      ) \
                      .sort("count", ascending=False)
```

与原始 SQL 相比，这种方法可以通过连续链接和测试操作，轻松迭代复杂查询，这是它的优势。此外，我们还可以使用 DataFrame API 访问大量的内置函数集合，比如之前使用的 `count`、`round` 和 `avg`

聚合函数。pyspark.sql.functions 模块还包含一些数学和统计工具，其中包含的函数可用于：

- 随机数据的生成
- 总结和描述性数据
- 样本协方差和相关性
- 交叉表（亦称列联表）
- 频率计算
- 数学函数

来使用一个这样的函数计算一些描述性的总结统计数据，以便对可用停车场地数据的分布和频率有更好的了解。describe 函数返回一个 DataFrame，其中包含每个指定数字列的非空条目计数、平均值、标准差、最小值和最大值：

```
parking.describe("regcap", "valetcap", "mccap").show()

+-------+------------------+------------------+------------------+
|summary|            regcap|          valetcap|             mccap|
+-------+------------------+------------------+------------------+
|  count|              1000|              1000|              1000|
|   mean|           137.294|             3.297|             0.184|
| stddev|361.05120902655824|22.624824279398823|1.9015151221485882|
|    min|                 0|                 0|                 0|
|    max|               998|                96|                 8|
+-------+------------------+------------------+------------------+
```

也许我们想确定停车场地所有者和停车场主要类型（“primetype”）之间的联合频率分布。在统计学中，这通常通过计算列联表或交叉表完成，这两种表以矩阵格式显示两个变量之间的共现频率。使用 stat 接口中的 crosstab 方法就可以轻松为 Spark DataFrame 计算该分布了：

```
parking.stat.crosstab("owner", "primetype").show()

+--------------------+---+---+---+---+---+
|     owner_primetype|PPA|PHO|CPO|CGO|   |
+--------------------+---+---+---+---+---+
|          Port of SF|  7|  7|  0|  4|  0|
|                SFPD|  0|  3|  0|  6|  0|
|               SFMTA| 42| 14|  0|  0|  0|
|GG Bridge Authority|  2|  0|  0|  0|  0|
|                SFSU|  2|  6|  0|  0|  0|
|                SFRA|  2|  0|  0|  0|  0|
..output truncated..
```

数据整理DataFrame

请注意，因为“owner”列看上去是高基数维度，因此结果被截断为前 20 行数据。虽然 Pandas 和 R 的用户应该能很好地理解 Spark DataFrame API 中的许多操作和功能，但由于 DataFrame 本质上的不可变性和分布式特性，所以它的一些有别于 Pandas/R 的地方应该被注意。例如，尽管 Spark 在加载时尽可能推断数据类型，但默认的回退类型（fallback type）还是字符串，这一点在 SF 停车示例中的“regcap”列有所体现。在 Pandas 中，可以通过选择该列并使用 `astype` 轻松转换该列值的类型：

```
parking['regcap'].astype(int)
```

但由于 DataFrame 实际上只是 RDD 的封装，是不可变的集合，因此需要执行几个步骤才能将此列转换为 `int` 类型。这种解决方法会根据现有列创建一个新列，将其值转换为正确的类型，最后删除旧列。为了保留列名，首先使用 `withColumnRenamed` 方法将现有列重命名为“regcap_old”，然后使用 `withColumn` 方法添加新的“regcap”列，该列包含 `regcap_old` 中转换类型后的值：**6**

> **6**虽然这里使用的是 Spark 的 `cast` 方法，但是从 Spark 1.4 起也可以使用 `astype`，它是 `cast` 方法的 Pandas 友好别名。

```
parking = parking.withColumnRenamed('regcap', 'regcap_old')
parking = parking.withColumn('regcap', parking['regcap_old'].cast('int'))
parking = parking.drop('regcap_old')
parking.printSchema()
```

因为其他数值列也需要进行这个转化，所以本着 DRY 的精神，来定义一个工具函数，为任意列和数据类型执行这种转换：

```
def convert_column(df, col, new_type):
    old_col = '%s_old' % col
    df = df.withColumnRenamed(col, old_col)
    df = df.withColumn(col, df[old_col].cast(new_type))
    df.drop(old_col)
    return df

parking = convert_column(parking, 'valetcap', 'int')
parking = convert_column(parking, 'mccap', 'int')
```

```
parking.printSchema()
```

不幸的是，这个函数不能处理“latitude”和“longitude”，因为它们实际上是“location_1”结构中的字段。我们可以进行一些改良，定义另一个参数为“location_1”结构类型的函数，使用 Google 的 Geocoding API（http://bit.ly/1r2xH5F）执行经纬度查找，以返回邻域名称。使用 `requests`（http://docs.python-requests.org/en/master/）库来发送请求：

```
import requests

def to_neighborhood(location):
    """
    使用Google的Geocoding API执行经纬度的逆向查找
    https://developers.google.com/maps/documentation/geocoding/intro#rever
    """
    name = 'N/A'
    lat = location.latitude

    long = location.longitude
    r = requests.get(
    'https://maps.googleapis.com/maps/api/geocode/json?latlng=%s,%s' %
    (lat, long))

    if r.status_code == 200:
        content = r.json()
        # results是匹配地址的列表
        places = content['results']
        neighborhoods = [p['formatted_address'] for p in places if
        'neighborhood' in p['types']]

        if neighborhoods:
            # 地址格式为Japantown, San Francisco, CA
            # 所以根据逗号分割，返回邻域名称
            name = neighborhoods[0].split(',')[0]

return name
```

`to_neighborhood` 函数接受一个 `location` 结构并返回一个字符串类型，但是如何在列表达式中使用这个函数呢？`pyspark.sql.functions` 模块提供了用于注册 UDF 的 `udf` 函数。通过向 UDF 传递一个可调用的 Python 函数和该函数的返回类型对应的 Spark SQL 数据类型，我们声明了一个内联 UDF；在这个示例中，返回的是一个字符串，所以使用 `pyspark.sql.types` 中的 `StringType` 数据类型。注册后，可以使用该 UDF 通过一个

withColumn 表达式重新格式化“location_1“列：

```
from pyspark.sql.functions import udf
from pyspark.sql.types import StringType

location_to_neighborhood=udf(to_neighborhood, StringType())

sfmta_parking = parking.filter(parking.owner == 'SFMTA') \
                        .select("location_1", "primetype", "landusetyp",
                        "garorlot", "regcap", "valetcap", "mccap") \
                        .withColumn("location_1",
                        location_to_neighborhood("location_1")) \
                        .sort("regcap", ascending=False)
sfmta_parking.show()

+-------------------+----------+----------+--------+------+--------+-----+
|         location_1|primetype|landusetyp|garorlot|regcap|valetcap|mccap|
+-------------------+----------+----------+--------+------+--------+-----+
|    South of Market|       PPA|          |       G|  2585|       0|   47|
|                N/A|       PPA|          |       G|  1865|       0|    0|
|Financial District|       PPA|          |       G|  1095|       0|    0|
|       Union Square|       PPA|          |       G|   985|       0|    0|
.. output truncated ..
```

在 Spark 本地模式下，由于 Python 的全局解释器锁（global interpreter lock，GIL，https://wiki.python.org/moin/GlobalInterpreterLock）的线程限制，我们无法并行化对 API 的 HTTP 请求。因此，在本地模式下使用此实用程序将需要串行运行整个 RDD，这可能需要相当长的时间。因此，为了让操作在合理的时间内完成，此示例将 DataFrame 过滤到了合适的大小。

如你所见，使用 Spark 的 DataFrame API 定义、注册 UDF 的过程比使用 Pig 和 Hive 容易得多。一旦注册，UDF 不仅可以被同一个 Spark 集群上的其他程序使用，也可以被通过 JDBC/ODBC 接口连接到 Spark SQL 上的 BI 工具使用。这使得 udf 函数轻松成为了 DataFrame API 提供的最强大的函数，因为它向 SQL 用户展现了应用高级计算或操作的无限可能性。本书没有涉及的内置功能和函数还有很多，它们的数量也会随着 Spark 版本的发布而不断增长。

要查看受 pyspark 的 Spark SQL 和 DataFrame API 支持的类和函数的最新列表，请参见官方 API 文档。Spark 开发新闻的另一个重要来源是 Databricks 公司（https://databricks.com/），由 Spark 创始人创立。Databricks 经常发布博文，描述所有新添加到 API 中的主要功

能，例如“Statistical and Mathematical Functions with DataFrame in Spark”（http://bit.ly/26B8HDd）。

8.3　小结

在本章中，我们了解了 Pig 如何大大简化了 MapReduce 数据流水线的构建过程。Pig 的主要使用场景是传统的 ETL 数据流水线过程，但它也是一种很好的工具，非常适用于执行临时分析，并从大批量数据中构建迭代处理或预测模型，当分析越来越复杂时尤其如此。

我们还介绍了 Spark SQL 模块和 DataFrame API。Spark 提供了内置集成，支持对结构化数据集的关系处理，并允许用户在单个编程环境中将关系处理和复杂的分析相结合。DataFrame 为 Hadoop 或 Spark 的 Python 程序员提供了广泛的、前所未有的分析可能性。推荐你阅读你偏爱的语言栈的 Spark 官方 DataFrame API 文档（http://spark.apache.org/docs/latest/sql-programming-guide.html），进一步探索 Spark 的 DataFrame API；并时刻关注 Spark 新闻（http://spark.apache.org/news/），留意未来发展。

第 9 章　机器学习

机器学习计算旨在从当前和历史数据中推导出预测模型。它作出了一个固有假设，即经历越多训练或获取越多经验，学习获得的算法将改进越多。通过从大数据集训练出来的模型，机器学习算法可以在非常小的领域实现非常好的预测效果。

因此，大多数机器学习算法都涉及大规模计算。出于这个原因，机器学习计算非常适用于 Spark 等分布式计算范式，利用大型训练集生成有意义的结果。本章将介绍 Spark 内置的机器学习库——Spark MLlib（http://spark.apache.org/docs/1.5.0/mllib-guide.html）。它由许多常见的学习算法和实用程序组成，比如分类、回归、聚类、协同过滤、降维以及一个新的“机器学习流水线”框架——spark.ml。spark.ml 提供了一套统一的高级 API，可以帮助用户创建和优化实际的机器学习流水线。[1]

> 1参见 Spark 的 Machine Learning Library (MLlib) Guide（http://spark.apache.org/docs/1.5.0/mllib-guide.html）。

9.1　使用Spark进行可扩展的机器学习

在第 4 章中，我们将 Spark 作为一个可在 Hadoop 集群上运行的内存分布式计算引擎进行了介绍。而且，Spark 平台还附带了几个使用 Spark 处理引擎的内置组件，来支持其他类型的分析工作，这些功能都受益于 Spark 的计算优化。本章将仔细研究 Spark 的内置机器学习库——MLlib。该库包含一套通用的统计和机器学习算法和实用程序，它们都被设计为能在集群中扩展。2

> 2《Spark 快速大数据分析》，Holden Karau 等人著。本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1558。

有些人可能对数据挖掘和机器学习的编程库很熟悉，比如 Python 的 Weka（http://www.cs.waikato.ac.nz/ml/weka/）或者 Scikit-Learn（http://scikit-learn.org）。虽然用这些库能游刃有余地应对可以在单个机器上处理的中小型数据集，但对于要求分布式存储和并行处理的大型数据集，我们不仅需要可以处理分布式数据集的计算引擎，还需要为并行平台设计的算法。Spark MLlib 仅仅包含并行算法，使用 Spark 的 RDD 操作跨节点并行应用操作。好在有许多机器学习技术和算法都非常适合并行化。但一定要记住，与 Spark API 一样，使用 Spark MLlib 时要注意创建数据（作为 RDD），并以分布式并行化的方式对数据进行操作。例如，对一个原始类型的小数据集（Python 字典或列表）调用 `parallelize()`，以便将其提供给集群中的所有节点。

Spark MLlib 包括一些统计和机器学习技术，比如采样、相关计算、假设检验等。而我们将主要关注 MLlib 的机器学习算法。这类算法基于训练数据寻找算法行为的数学最优解，从而作出预测和决策。3Spark MLlib 学习算法集中在机器学习的三个关键领域，通常被称为机器学习的 3C。

> 3SNN Adaptive Intelligence,“What Is Machine Learning?”(http://www.mlplatform.nl/what-is-machine-learning/).

协同过滤（collaborative filtering）

　　也被称为推荐引擎。它基于过去的行为、偏好或与已知实体 / 用户的相似性产生推荐。

分类（classification）

也被称为有监督学习。它从监督训练集中学习，并根据该训练集对未分类项进行分类。

聚类（clustering）

也被称为无监督学习。它基于类似特征，将数据分组为集群。

一般来说，要想实现这些算法，得先从数据中定义和提取一组特征作为特征的数值表示。例如，如果我们要设计一个能推荐具有相似属性（价格、颜色、品牌等）产品的推荐系统，就可以定义由每个产品属性的加权数值组成的特征向量。或者，当我们想提取非结构化文本的特征（即基于垃圾邮件，检测过滤电子邮件）时，则可以用每个词在每个分类类别（即是垃圾邮件或不是垃圾邮件）的词频 - 逆文档频率（TF-IDF）的向量来表示它。

一旦从数据中提取了特征向量，就可以将它们作为训练数据提供给机器学习算法，该算法将返回表示预测的训练模型。在训练有监督学习模型时，通常会保留一部分训练数据作为“测试数据”，将模型应用于测试数据，并通过比较测试数据的预测结果与实际结果来量化模型的准确性。这使我们能评估模型的准确性并优化其精度。机器学习流水线的概览图如图 9-1 所示。

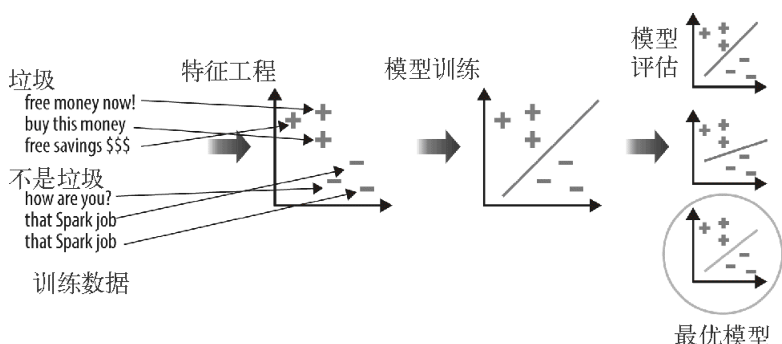


图 **9-1**：机器学习流水线

我们将把这种常见的机器学习流水线应用于接下来的几个例子中，并使用 MLlib 内置的一些评估工具来评估我们的学习模型。假设你已经安装了 Spark 并符合运行 Spark MLlib 的要求，具体内容请参见附录 B。

9.1.1 协同过滤

协同过滤（或推荐系统）应该最常见于电子商务领域，比如亚马逊和 Netflix 等公司通过挖掘用户行为数据（如浏览、评分、点击和购买）来推荐其他产品。广义上，协同过滤算法分为两种。

基于用户的推荐系统

查找与目标用户相似的用户，使用其协同评分为目标用户提供推荐。

基于物品的推荐系统

查找并推荐与目标用户相关联物品相似或相关的物品。

MLlib 的协同过滤库专注于基于用户的推荐，使用交替最小二乘法（alternating least squares，ALS）算法的实现。4MLlib 的协同过滤方法将用户的偏好表示为用户物品关联矩阵，其中每位用户和物品的点积通过将偏好评分（或评级）与加权因子相乘获取。这样，我们就能接收用户的显式反馈（例如正评分、购买）和隐式反馈（例如浏览、点击），并将它们并入模型中，作为二元偏好和置信度值的组合。然后，该模型将去寻找可用于预测物品的预期偏好的隐语义因子。

> 4Koren Yehuda et al.,“Matrix Factorization Techniques For Recommender Systems”(http://dl.acm.org/citation.cfm?id=1608614), Computer 14.8(2009):30–37.

示例：一个基于用户的推荐系统

试着用 MLlib 的 ALS 算法为在线约会服务生成推荐（或潜在）对象。我们将根据已有的交友网站的个人资料评分数据集为特定用户生成推荐内容。

在 GitHub 仓库的 data/mllib/dating 目录中，你可以找到包含数据集的两个 CSV 文件：168 791 份用户个人资料（gender.dat），以及一百万条以上关于他们的用户评分数据（ratings.dat）。这些数据可从 Occam 的实验室（http://www.occamslab.com/petricek/data）获取。

评分数据遵从以下格式：UserID、ProfileID、Rating。UserID 是提供评分的用户，ProfileID 是被评分的用户，Rating 是 1~10 的评分，其中 10 是最高分。

UserID 的范围介于 1～135 359，ProfileID 的范围介于 1～220 970（并非每份用户资料都被评过分）。只有至少提供了 20 条评分的用户才会被纳入，一直打同样分数的用户将被排除。

用户的性别信息遵从以下格式：UserID、Gender，其中男性为“M”、女性为“F”、未知为“U”。

可运行的完整约会推荐程序可以在 GitHub 仓库的如下目录中找到：

```
hadoop-fundamentals/mllib/collaborative_filtering/als/matchmaker.py
```

使用 `spark-submit` 命令，并向它传递两个参数：UserID（要为谁生成推荐）以及 M 或 F（对伴侣的性别偏好），该程序就可以在 Spark 上运行了。建议将此输出通过管道写入到一个文件中：

```
$ $SPARK_HOME/bin/spark-submit \
~/hadoop-fundamentals/mllib/collaborative_filtering/als/matchmaker.py 1 M
> ~/matchmaking_recs.txt
```

我们将分析该程序的每个主要步骤。首先，使用应用程序的名称配置 SparkContext，并将每个 executor 使用的内存大小设置为 2GB，因为 ALS 算法处理这个数据量需要大量内存：

```
# 配置Spark
conf = SparkConf().setMaster("local") \
                  .setAppName("Dating Recommender") \
                  .set("spark.executor.memory", "2g")
sc = SparkContext(conf=conf)
```

接下来，读取 UserID 参数以及用户的性取向，并对评分文件中的每条记录调用自定义的 `parse_rating` 方法：

```
def parse_rating(line, sep=','):
    """
    解析评分行
    返回：元组(随机整数, (user_id, profile_id, rating))
    """
    fields = line.strip().split(sep)
    user_id = int(fields[0]) # 将user_id转换为int
    profile_id = int(fields[1]) # 将profile_id转换为int
    rating = float(fields[2]) # 将rated_id转换为int
    return random.randint(1, 10), (user_id, profile_id, rating)
```

```
```

给定一个评分行，parse_rating 方法返回一个元组，其中第一项是一个随机整数，第二项是另一个元组 (user_id, profile_id, rating)：

```
matchseeker = int(sys.argv[1])
gender_filter = sys.argv[2]

# 创建评分RDD (随机整数, (user_id, profile_id, rating))
ratings = sc.textFile(
    "/home/hadoop/hadoop-fundamentals/data/dating/ratings.dat")\
    .map(parse_rating)
```

为元组中的第一项生成一个随机数，之后它将作为键将此 RDD 分解为训练集和测试集。ALS 要求将 Rating 对象表示为 (UserId, ItemId, Rating) 元组。在这个示例中，ItemId 实际将映射到其他用户资料的用户 ID。

接下来，通过将自定义的 parse_user 方法映射到 gender.dat 的每一行，读取用户个人资料数据：

```
def parse_user(line, sep=','):
    """
    解析用户行
    返回：元组(user_id, gender)
    """
    fields = line.strip().split(sep)
    user_id = int(fields[0]) # 将user_id转换为int
    gender = fields[1]
    return user_id, gender
```

给定一个用户行，parse_user 方法返回一个元组 (user_id, gender)。一旦用户元组的 RDD 生成，就调用 collect() 将 RDD 转换为列表：

```
# 创建用户RDD
users = dict(sc.textFile(
"/home/hadoop/hadoop-fundamentals/data/dating/gender.dat")\
.map(parse_user).collect())
```

现在来将评分数据分为训练集和验证集，训练集用于训练模型，验证

集用于评估模型。通过对添加到每个元组的随机整数键进行过滤，保留 60%的训练数据和 40%的验证数据。将分区数设置为 4（或计算机支持的处理器内核数）并缓存结果，提高 RDD 的并行性。

```
# 基于时间戳的最后一位，创建训练集（60%）和验证集（40%）
num_partitions = 4
training = ratings.filter(lambda x: x[0] < 6) \
                  .values() \
                  .repartition(num_partitions) \
                  .cache()

validation = ratings.filter(lambda x: x[0] >= 6) \
                    .values() \
                    .repartition(num_partitions) \
                    .cache()

num_training = training.count()
num_validation = validation.count()

print "Training: %d and validation: %d\n" % (num_training, num_validation)
```

可以通过设置和调整 ALS 提供的这些训练参数来优化模型。

`rank`

使用的特征向量的大小，由隐语义因子的数量决定；`rank` 越大，产生的模型越好，但计算的代价也更大（默认值为 10）。

`num_iterations`

迭代次数（默认值为 10）。

`lambda`

正则化参数（默认值为 0.01）。

`alpha`

常数（默认值为 1.0），用于计算隐式反馈 ALS 的置信度。

因为此例只捕获显式评分，所以将忽略 `alpha` 并使用默认值。其他参数中，`rank` 使用 8，将迭代次数设置为 8，`lambda` 为 0.1。由于对数据还不够了解，无法确定隐语义因子的数量或合适的正则化值，所以这些初始训练参数设置得有些随意。然而，可以先从这个组合开始，将它与使用其他训练参数组合的模型进行对比来评估结果，确定

最佳拟合模型：

```
# rank是模型中隐语义因子的数量
# num_iterations是迭代次数
# lambda指定ALS中的正则化参数
rank = 8
num_iterations = 8
lambda = 0.1
```

现在使用 `ALS.train()` 方法创建模型，该方法接受评分元组的训练 RDD 和我们的训练参数：

```
# 使用训练数据、已配置的rank和迭代参数训练模型
model = ALS.train(training, rank, num_iterations, lambda)

# 使用验证集评估经过训练的模型
print "The model was trained with rank = %d, lambda = %.1f, and %d iterati
\n" % \
    (rank, lambda, num_iterations)
```

在详细日志记录模式下运行 `train()` 方法时要小心，此操作需要进行几次 RDD 投影和操作，因此可能会有长达几分钟的日志滚动。

模型一旦被创建，就使用均方根误差（root mean squared error，RMSE）来计算每个模型的误差。RMSE 是拥有实际评分的所有用户的（实际评分 - 预测评级）^2 的平均值的平方根。**5**

5Kaggle,“Root Mean Squared Error”(https://www.kaggle.com/wiki/RootMeanSquaredError).

$$RMS = \left(\frac{1}{n} \sum_{i=1}^{n} (\text{model}_{\ i} - \text{observed}_{\ i})^2 \right)^{\frac{1}{2}}$$

我们的推荐程序也可以相应地实现 RMSE 计算：

```
def compute_rmse(model, data, n):
    """
    计算RMSE，或者(实际评分-预测评分)^2的平均值的平方根
    """
    predictions = model.predictAll(data.map(lambda x: (x[0], x[1])))
    predictions_ratings = predictions.map(lambda x: ((x[0], x[1]), x[2]))
```

```
    .join(data.map(lambda x: ((x[0], x[1]), x[2]))) \
    .values()
  return sqrt(predictions_ratings.map(lambda x: (x[0] - x[1]) ** 2). \
  reduce(add) / float(n))
```

RMSE 表示模型对数据的绝对拟合（观察数据点与模型预测值的接近程度），并且与评分值的单位相同。RMSE 值越小，拟合度越高。但由于它与评分值相关，所以应该按 1~10 进行评估。根据结果，可以通过调整训练参数，或者提供更多或更好的训练数据，来优化模型：

```
# 打印模型的RMSE
validation_rmse = compute_rmse(model, validation, num_validation)

print "The model was trained with rank=%d, lambda=%.1f, and %d iterations.
(rank, lambda, num_iterations)
print "Its RMSE on the validation set is %f.\n" % validation_rmse
```

假设我们对 RMSE 值反映出来的模型拟合程度很满意，就可以用它来为给定用户生成推荐了。通过根据给定用户的性取向进行过滤，先生成一组符合条件的用户。这是推荐候选人 RDD：

```
# 根据性取向进行过滤
partners = sc.parallelize([u[0] for u in filter(lambda u: u[1] ==
 gender_filter, users.items())])
```

现在使用模型的 `predictAll()` 方法，向它传递一个二元元组 RDD，RDD 的 key 是 `user_id`，其中 `user_id` 是给定的要为之生成推荐的用户（matchseeker）。这样，模型就能生成推荐。我们会将结果收集到一个列表中，按照评分值降序排序，取前 10 名推荐用户：

```
# 使用经过训练的模型进行预测
predictions = model.predictAll(partners.map(lambda x: (matchseeker, x))) \
.collect()

# 对推荐进行排序
recommendations = sorted(predictions, key=lambda x: x[2], reverse=True)[:1
```

最后，打印完整的推荐列表并停止 SparkContext：

```
print "Eligible partners recommended for User ID: %d" % matchseeker
```

```
for i in xrange(len(recommendations)):
    print ("%2d: %s" % (i + 1, recommendations[i][1])).encode('ascii', 'ig

# 清理
sc.stop()
```

如果使用前面的命令将此作业提交到 Spark，并将输出保存到一个结果文件中，应该会看到类似于以下内容的输出：

```
$ cat matchmaking_recs.txt

Training: 542953 and validation: 542279

The model was trained with rank = 8, lambda = 0.1, and 8 iterations.

Its RMSE on the validation set is 3.580347.

Eligible partners recommended for User ID: 1
 1: 100939
 2: 70020
 3: 109013
 4: 54998
 5: 132170
 6: 3843
 7: 170778
 8: 51378
 9: 8849
10: 118595
```

RMSE 是评估模型性能的重要指标。与大多数机器学习算法一样，协同过滤模型中的数据和迭代次数越多，模型表现越好。建议你尝试对 ALS 使用不同的 `rank`、迭代次数和正则化（`lambda`）参数的组合，比较 RMSE 以找到最佳拟合的参数组合。你可以在 Spark MLlib 文档的“Collaborative Filtering”（http://spark.apache.org/docs/latest/mllib-collaborative-filtering.html#scaling-of-the-regularization-parameter）找到更多有关参数调优的信息，以及一个实现电影推荐系统的 ALS 算法示例。

9.1.2　分类

分类试图通过有监督训练方法将数据（通常是文本或文档）分类，这些方法使用带标注的训练集来发现模式，使机器学习程序能快速标记新记录。例如，一个简单的分类算法可能会记录与类别相关联的特征和词，以及该词出现在给定类别中的次数。一旦机器学习程序从训练

数据中提取出了特征，它就可以生成特征向量并应用统计模型构建预测模型，然后将预测模型应用于新的数据。

MLlib 提供了一些用于二分类、多分类以及回归分析的算法。在二分类中，我们希望将实体分为两个不同的类别或标签（例如确定电子邮件是否为垃圾邮件）；在多分类中，我们希望将实体分类为两个以上的类别（例如确定新闻报道属于哪个类别）；回归分析算法的目标是，使用连续函数估计因变量（例如身体活动水平）与一个或多个自变量（例如心脏病的风险）之间的关系和依赖性。

在这些类型的算法中，MLlib 实现都要对一组带标签的例子（example）应用算法。这些例子被表示为 `LabeledPoint` 对象，包括一个数值（用于二分类）或特征向量（用于多分类）以及类别标签。已经分类的 `LabeledPoints` 中的训练数据用于训练模型，然后用模型去预测新实体的类别。

MLlib 的官方文档（http://bit.ly/26Bp4zE）按类别列举了所有支持的分类算法。本节将通过随机梯度下降的逻辑回归过程（也被称为 `LogisticRegressionWithSGD`）创建一个简单的二分类器。

示例：一个逻辑回归分类

在这个示例中，我们将构建一个简单的垃圾邮件分类器，使用已经分类（垃圾邮件和非垃圾邮件）的电子邮件数据进行训练。此垃圾邮件分类器将使用两个 MLlib 算法：`HashingTF` 和 `LogisticRegressionWithSGD`，前者从训练文本提取词频向量作为特征向量，后者使用随机梯度下降（http://bit.ly/26Bp7vf）实现逻辑回归。

可以在 GitHub 仓库的 /data 目录中的 spam_classifier.zip 中找到训练数据 spam.txt 和 ham.txt。此数据是 SpamAssassin 公共语料库（http://spamassassin.apache.org/publiccorpus/）的一个子集。完整的垃圾邮件分类程序可以在 mllib/classification 目录下找到，可以使用以下命令运行程序：

```
$ $SPARK_HOME/bin/spark-submit \
/home/hadoop/hadoop-fundamentals/mllib/classification/spam_classifier.py \
/home/hadoop/hadoop-fundamentals/data/spam_classifier/spam.txt \
/home/hadoop/hadoop-fundamentals/data/spam_classifier/ham.txt
```

我们将讲解每个主要步骤。

首先配置 SparkContext，设置应用程序名称，并将 executor 内存增加到 2GB：

```
# 配置Spark
conf = SparkConf().setMaster("local") \
                  .setAppName("Spam Classifier") \
                  .set("spark.executor.memory", "2g")
sc = SparkContext(conf=conf)
```

接下来读取命令行参数，获取训练数据文件的路径。读取这些文件，创建垃圾邮件（spam）和非垃圾邮件（ham）RDD：

```
spam_file = sys.argv[1]
ham_file = sys.argv[2]

spam = sc.textFile(spam_file)
ham = sc.textFile(ham_file)
```

现在，实例化 `HashingTF` 对象，将要提取的特征数量设置为 10 000：

```
tf = HashingTF(numFeatures=10000)
```

将 `HashingTF` 的 `transform()` 方法应用于 spam 和 ham 数据，首先将内容分成单词令牌。这将从 spam 和 ham RDD 中提取出词频向量，并将其投影为新的特征向量 RDD：

```
spam_features = spam.map(lambda email: tf.transform(email. \
 split(" ")))
ham_features = ham.map(lambda email: tf.transform(email. \
 split(" ")))
```

现在将 RDD 中的每个特征向量转换为 `LabeledPoint`。因为这是一个二分类器，因此用 1 表示垃圾邮件，用 0 表示非垃圾邮件。`LabeledPoint` 对象的第二个值将包含该特征。将这些 RDD 的并集作为训练数据集并缓存它，因为逻辑回归是一种迭代算法：

```
positive_examples = spam_features.map(lambda features: LabeledPoint(1, fea
negative_examples = ham_features.map(lambda features: LabeledPoint(0, feat
training = positive_examples.union(negative_examples)
training.cache()
```

现在使用 SGD 算法和训练数据运行逻辑回归：

```
model = LogisticRegressionWithSGD.train(training)
```

现在创建测试数据，包括应分类为垃圾邮件的文本内容，以及应分类为非垃圾邮件的文本内容。然后使用训练模型来预测测试数据是否被视为垃圾邮件。回想之前对 `LabeledPoints` 的设置，1 表示垃圾邮件，0 表示非垃圾邮件：

```
# 创建测试数据，测试模型
positive_test = tf.transform("Guaranteed to Lose 20 lbs in 10 days
Try FREE!".split(" "))
negative_test = tf.transform("Hi, Mom, I'm learning all about Hadoop
and Spark!".split(" "))

print "Prediction for positive test example: %g" % model.predict(positive_
print "Prediction for negative test example: %g" % model.predict(negative_
```

至此，就可以将预测结果与数据的分类进行比较，评估分类器模型的准确性，或者将该模型应用于未标记的数据集。MLlib 的分类算法针对大型监督训练数据进行了优化。因此，比起少而精，更多的数据通常会产生更好的效果。然而，分析数据并应用最合适的算法和评估方法仍然很重要。请参考官方的 Spark MLlib 文档（http://spark.apache.org/docs/latest/ml-guide.html），了解所有支持的分类算法和评估指标，特别要注意它们各自支持的 API（Scala、Java、Python）。

9.1.3　聚类

与协同过滤和分类算法不同，聚类利用无监督学习技术来构建模型。聚类算法尝试将数据集合组织成类似项目的分组，比如寻找具有相似特征或兴趣的客户群体，或将动植物按常见物种分组。聚类的目标是将数据分成多个簇，使每个簇内的数据彼此之间比与其他簇中的数据更相似。6

> 6《Hadoop 硬实战》，Alex Holmes 著。

Spark MLlib 提供了一些流行的聚类模型，但其中最简单也是最流行

的聚类算法恐怕还得属 k-means。k-means 算法需要将所有对象表示为一组数值特征，并事先指定想要的目标簇数（k 个簇）。

MLlib 的 k-means 聚类的实现也从向量化数据集开始，将每个对象表示为 n 维空间中的特征向量，其中 n 用于描述要聚类的对象的所有特征的数量。算法首先在该向量空间随机选择 k 个点作为簇的初始中心或质心，然后将每个对象分配给最接近的质心，使用簇中所有点的坐标的平均值重新计算质心点，并根据需要将对象重新分配到最近的簇。分配对象和重新计算中心的过程不断重复，直到过程收敛，如图 9-2 所示。7

> 7《Spark 快速大数据分析》Holden Karau 等人著。本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1558。

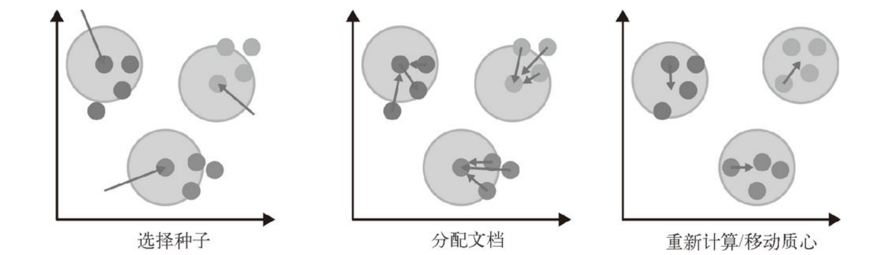


图 **9-2**：***k*-means** 聚类算法的计算阶段

聚类中最重要的问题就是确定如何量化要聚类的对象的相似度。加权方法可以从 TF-IDF 获得，这对于文本文档特别有用；另一种加权方法是通过数据中的其他自定义属性（使用计算指标衡量的平均占比）的函数来确定（例如基于以美元为单位的总购买金额划分顾客）。对于 MLlib 的 k-means 聚类的输入，需要指出对特征向量使用的加权方法。例如，如果确定要根据总购买金额、平均购买频率和每次平均购买金额这三个特征对所有客户进行聚类，那么客户样本可能如表 9-1 所示。

表**9-1**：客户特征向量化

		[illegible]		
[illegible],5,115]				
[illegible],1,45]				
[illegible],2,65]				

有多个特征时一定要注意，维度值是以不同单位表示的，或者尚未进行归一化。如果使用简单的、基于距离的指标来确定这些向量之间的相似性，总购买金额将主导结果。通过给不同的维度加权，可以解决这个问题。**8**

> **8**《Mahout 实战》，Sean Owen、Robin Anil、Ted Dunning、Ellen Friedman 合著。

示例：一个*k*-means聚类

在这个例子中，我们将应用 *k*-means 聚类算法来确定截至今年美国哪些区域发生地震的次数最多。**9** 这些信息可以在 GitHub 仓库的 /data 目录中的 earthquakes.csv 文件中找到。此 CSV 文件的列如下所示：

> **9**参见“USGS Earthquakes Hazard Program”（http://earthquake.usgs.gov/earthquakes/feed/v1.0/csv.php）。

- time
- latitude
- longitude
- depth
- magnitude
- magnitudeType
- nst
- gap
- dmin
- rms
- net
- jd
- updated
- place

从这些记录中提取纬度（latitude）和经度（longitude），并用其训练模型。在这个迭代过程中，我们将尝试生成 6 个簇。完整的程序可以使用以下命令运行：

```
$ $SPARK_HOME/bin/spark-submit \
/home/hadoop/hadoop-fundamentals/mllib/clustering/earthquakes_clustering.p
/home/hadoop/hadoop-fundamentals/data/earthquakes.csv \
6 > clusters.txt
```

首先，配置 Spark 并创建 SparkContext：

```
# 配置Spark
conf = SparkConf().setMaster("local") \
                  .setAppName("Earthquake Clustering") \
                  .set("spark.executor.memory", "2g")
sc = SparkContext(conf=conf)
```

接下来，从地震数据文件创建训练 RDD，解析每一行的纬度和经度，并将其转化为一个 NumPy 数组：

```
# 创建用于训练的(lat, long) RDD向量
earthquakes_file = sys.argv[1]
training = sc.textFile(earthquakes_file).map(parse_vector)
```

使用第二个参数设置 *k* 个簇，在本示例中为 6：

```
k = int(sys.argv[2])
```

调用 `KMeans.train()`，将训练集和 *k*（设置为 6）传递给它。这将生成模型，我们也可以访问簇的中心：

```
# 基于训练数据和k-clusters训练模型
model = KMeans.train(training, k)

print "Earthquake cluster centers: " + str(model.clusterCenters)
sc.stop()
```

如果检查输出的 clusters.txt 文件，就会看到类似如下的输出：

```
Earthquake cluster centers: [array([ 38.63343185, -119.22434212]),
array([ 13.9684592 , 142.97677391]),
array([ 61.00245376, -152.27632577]),
array([ 35.74366346, 27.33590769]),
array([ 10.8458037, -158.656725 ]),
array([ 23.48432962, -82.3864285 ])]
```

现在，就可以根据训练数据绘制结果输出，对结果执行"眼球"评估，并通过优化簇数（*k*）和迭代次数来调整簇中心了。为了获取更精确的评估指标，还可以计算"集内平方误差的总和"（Within Set Sum of

Squared Errors），它衡量每个中心点周围簇点的紧凑度：

```
def error(point):
    center = model.centers[model.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))

WSSSE = training.map(lambda point: error(point)).reduce(lambda x, y: x + y
print("Within Set Sum of Squared Error = " + str(WSSSE))
```

9.2　小结

本章实现了一个基于用户的简单推荐系统，使用实现逻辑回归的二分类器对电子邮件进行了分类，使用 *k*-means 聚类算法对文档集合进行了聚类，并介绍了一点输入数据的向量表示。但是，这只是 MLlib 的预测分析能力的皮毛。

除了其他算法和数据准备工具，MLlib 还提供了评估算法的质量和性能的工具。希望这个简短的介绍展示了 MLlib 将强大的统计学习技术应用于大型数据集的潜力。由于 Spark MLlib 正逐步发展成一个更广泛的分布式机器学习框架，因此我们也鼓励你去深入了解它的统计和机器学习能力以及未来发展情况。数据类型、算法和实用程序都可以在官方的 Spark MLlib 指南（http://spark.apache.org/docs/latest/mllib-guide.html）中找到。我们还推荐你阅读 Sandy Ryza、Uri Laserson、Sean Owen 和 Josh Wills 合著的《Spark 高级数据分析》[10]，这是一本以示例驱动的优秀图书。

[10] 本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1668。

第 10 章　总结：分布式数据科学实战

在本书中，我们领略了 Hadoop 生态系统的具体组成部分。第一部分讨论了如何与集群进行交互以及如何使用集群。如前所述，Hadoop 是一种分布式计算的操作系统；和本地计算机上提供文件系统和进程管理的操作系统一样，Hadoop 通过 HDFS 以及 YARN 形式的资源和

调度框架提供分布式数据存储和访问。HDFS 和 YARN 一起构成了一种在极大数据集上进行分布式分析的机制。

编写分布式作业的原始方法是使用 MapReduce 框架，它让你能指定 mapper 和 reducer 任务。将这些任务链接在一起，可以进行更庞大的计算。由于 Python 是数据科学最流行的工具之一，所以本书专门探讨了如何通过 Hadoop Streaming 执行使用 Python 脚本的 MapReduce 作业。在第一部分中，还探索了一种更纯粹的解决方案：使用 Spark 的 Python API 在使用了 YARN 的 Hadoop 集群中执行 Spark 作业。最后，介绍了在集群上常用的分布式分析和设计模式，结束了对低级工具的讨论。

第二部分从低级编程细节完全转移到了数据挖掘、数据采集、数据流和机器学习所用的高级工具上。这一部分主要针对通过 Hadoop 的各种现有工具进行分布式数据分析的日常情形，以及如何根据大数据流水线（数据采集、数据整理 /staging、计算和分析、工作流管理）组织这些工具。

你可能会有一个疑问：如何将 Hadoop 和 Spark 中的所有这些工具、组件组合在一起？

在第 1 章中，我们讨论了为什么大数据变得重要了——主要由于数据产品（从数据中获取价值，并通过应用预测或模式识别来生成新数据的应用程序）的兴起。数据产品必须具有自适应性和广泛适用性（可通用）；因此，机器学习和强化技术在成功部署数据产品方面越来越突出。数据产品的自适应行为要求它们不能是静态的，而是要不断学习；可通用性需要大量的数据参考点来拟合模型。因此，数据产品需要分布式计算来处理多样的、快速的数据——这也正是现代机器学习的特点。

> 数据产品是构建的消费品（不一定完全是软件），它从数据中获取价值并生成新数据。要实现该定义，必然需要应用机器学习技术。数据驱动的应用程序只是使用数据的应用程序（包括每个软件产品），例如博客、网上银行、电子商务等。即使数据驱动的应用程序从数据中获取了价值，它也不一定会生成新的数据。

本章将详细介绍如何使用本书中讨论过的所有工具来构建数据产品，并在此过程中，回答如何将分布式计算的低级操作和高级生态系统工具拟合在一起。即便本书只是 Hadoop 和分布式计算的一个入门介

绍，但我们也想在总结时提供一些建议，看看接下来能做什么。希望通过将整个数据产品和机器学习生命周期进行语境化，你能更轻松地识别和了解对工作流至关重要的工具和技术。

10.1 数据产品生命周期

构建数据产品需要建立和维护活动的数据工程流水线。流水线包括采集、整理、仓储、计算和探索性分析等多个步骤，这些步骤一同构成了数据工作流管理系统。它的主要目标是建立和实施拟合的（经过训练的）模型，其核心过程包括提取、转换和加载（ETL）过程——从应用程序上下文中提取数据，将其加载到 Hadoop 中，在 Hadoop 集群中处理数据，然后将数据 ETL 回应用程序。如图 10-1 所示，可以将这个简单的流程图看作是一个活动的或者常规的生命周期。在这个周期内，通过新的数据和交互，为用户调整和使用机器学习模型。

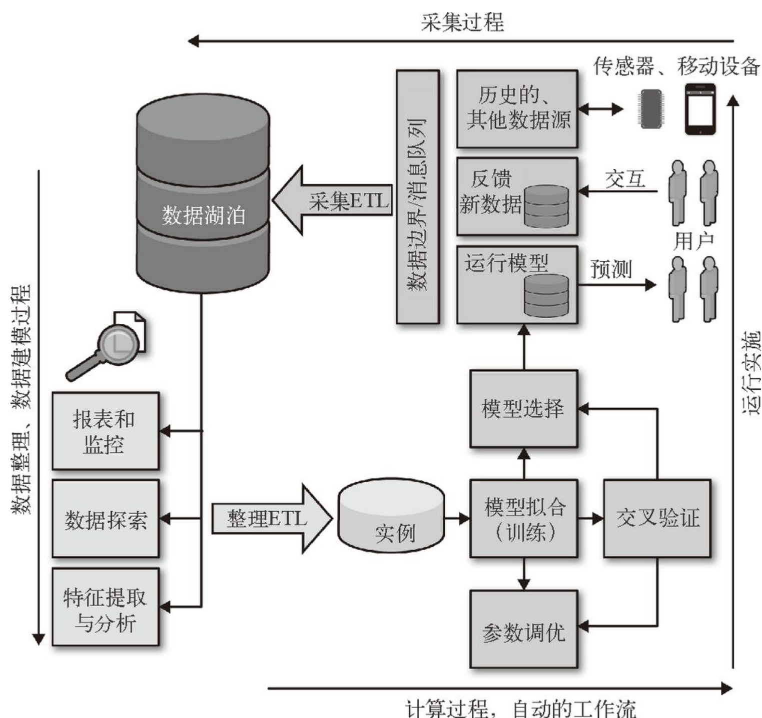


图 10-1：数据产品生命周期

如果想充分运用机器学习算法，数据产品生命周期就需要使用大数据分析和 Hadoop。拥有大量用户的应用程序必然将产生大量数据。尽

管通过 128GB 内存和多个内核的强大服务器进行有效的抽样和分析后，这些数据可以被处理，但数据的多样和高速才是需要 Hadoop 和基于集群的方法的灵活性的主要原因。

灵活性确实是基于集群的系统的关键。Web 日志记录（点击流式数据）、用户交互和流式数据集（例如传感器数据）形式的输入数据源源不断地涌入应用程序。这些数据源被写入各处，比如日志文件、NoSQL 数据库，以及 Web API 后端的关系数据库。此外，来自网络爬取、数据服务和 API、调查及其他业务来源的数据等信息也不断被生成。这些额外的数据必须与现有应用程序数据被一道分析，从而确定能改进数据产品模型的特征是否存在。

因此，数据产品生命周期通常围绕一个或多个中心数据存储。数据存储极其灵活，没有约束（不像关系数据库中存在约束），但持久性强。这样的中心数据存储是 WORM 系统，即“写一次，读多次”，是向下游分析提供可靠数据的关键所在，支持历史分析和可重复的 ETL 生成（这对科学至关重要）。WORM 存储系统对数据科学非常重要，它们也因此收获了一个新名字——数据湖泊。

10.1.1 数据湖泊

传统上，我们会使用数据仓库模型来执行业务环境中的常规聚合分析。数据仓库是关系数据库的扩展，通常将数据规一化为星型模式——将多个维表连接到一个中心事实表（所以关系图看起来像星型）的模式。事务通常发生在维表上，它们的解耦使组织的各个方面在读写上有了一些性能优势。ETL 过程通过一个大连接构建“数据（超）立方”来加载事实表，我们可以在数据立方上应用枢纽分析（pivot）和其他分析方法。

为了有效利用传统的数据仓库，必须先设计一个清晰的模式，经历冗长周期的 ETL 过程，完成数据库管理、数据转换和加载后，才可以分析数据。不幸的是，当你将数据产品视为需要新数据和新数据源的、有生命的、活动的引擎时，这种传统的数据分析模型可能既费时又有限制。对应用程序的简单改动（例如新的历史数据源，或新的日志记录和提取技术）就可能改变数据立方的结构，需要重新设计星型模式的范式。这种结构调整不仅费时费力，而且还迫使我们作出一个业务决策：是否值得为这些数据增加机器，来处理新的数据量？

作为数据科学家，我们都知道所有数据至少有潜在价值，但却很难回答数据价值及其相对成本效益的问题。因此，许多公司不仅在数据仓库上有所投入，而且还开发数据湖泊作为主要的数据收集和同步策

略。

数据湖泊支持从各种源（结构化和非结构化的）中流入未处理的原始数据，它将整个数据集合存储在一起，无须太多的组织，如图 10-2 所示。结构化数据可以从关系数据库、结构化文件（如 XML 或 JSON）和符号分隔文件（如日志文件）中获取，并且通常以基于文本的格式或某种类型的序列化二进制格式（如 SequenceFiles、Avro 或 Parquet）加载到系统。半结构化和非结构化数据包括传感器数据、二进制数据（如图像）和不是面向记录而是面向文档的文本文件（如电子邮件）。数据湖泊模式允许任何类型的数据自由流入存储，然后通过在处理时施加所需模式的 ETL 过程流出。根据分析需求进行提取和转换后，数据被加载到一个或多个数据仓库进行例行分析或紧急分析。数据湖泊模式提供了在线访问整套原始的源格式数据的功能，并将模式定义延迟到处理时，让公司能在需求变化时敏捷地执行新的处理和分析。

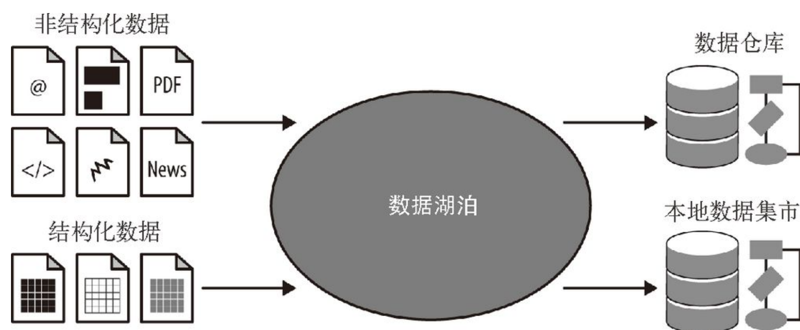


图 **10-2**：结构化和非结构化的数据流入数据湖泊，后者使用 **ETL** 过程查询，产生一个可以分析的数据仓库

尽管本书专注于 HDFS，但还有其他许多分布式数据存储解决方案，比如 GlusterFS、EMC 的 Isilon OneFS 和 Amazon 的 Simple Storage Service（S3），等等。然而，HDFS 是 Hadoop 的默认文件系统，也是构建数据湖泊非常有效的方式。HDFS 将数据分发到许多机器，支持用许多小容量的硬盘存储数据；同时，使数据可用于分布式框架中的计算，而不会产生存储区域网络（storage area network，SAN）的网络流量。此外，HDFS 会复制数据块，提供持久性和容错性，确保数据不会丢失。NameNode 以分层文件系统的形式组织数据命名空间，而不设计每个字段数据的模式。

与其使用负载过重又容量有限的单个主数据仓库，还不如将数据存储在 HDFS 数据湖泊中，通过 MapReduce 或 Spark 作业进行灵活分

析，然后被提取并加载到目标系统中，例如需要特定类型分析的业务部门的企业数据仓库。此外，通常可以将在磁带上存档的、无法分析的旧历史数据转移到 Hadoop 上，进行探索性分析。如此看来，Hadoop 可以减轻传统数据仓库的沉重维护负担、突破后者在可扩展性上的限制，甚至补充了现有的数据仓库架构，如图 10-3 所示。

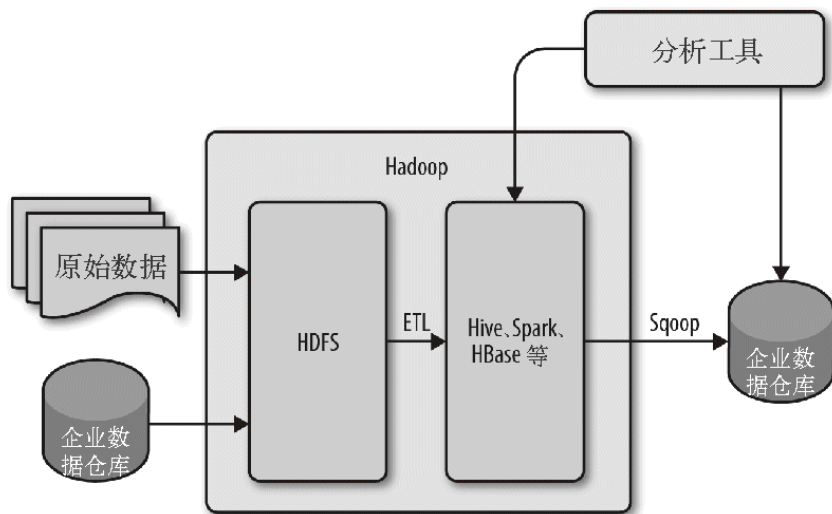


图 **10-3**：使用 **Hadoop** 的混合数据仓库架构

10.1.2　数据采集

深入了解数据产品生命周期的中心对象——数据湖泊之后，就可以将注意力转移到数据采集和数据仓储，以及数据科学家通常如何看待这些流程了。首先从数据采集开始。

一般来说，大多数数据采集从应用程序上下文获取数据，也就是和用户交互的软件产品的业务单元，或者实时收集信息的逻辑单元。例如，一个规模可观的电子商务平台可能有一个只处理客户评论的软件应用程序，和一个只收集用于安全和日志的网络流量信息的单元。这两个数据源对于异常检测（欺诈）或推荐系统等数据产品非常有价值，但必须分别采集进入数据湖泊。本节将介绍 Sqoop 和 Flume 这两种工具，它们都支持这两种上下文采集。

Sqoop 可以利用 JDBC（Java database connector）库连接到任何关系数据库系统，并将其导出到 HDFS。关系数据库几乎是目前所有 Web 应用程序和串行（非分布式）分析的后端服务器。由于关系数据

库是小规模分析的着力点，在 Web 应用程序中无处不在，因此 Sqoop 是将数据从大多数大型数据源提取到 HDFS 中的重要工具。此外，由于 Sqoop 从关系上下文中提取数据，因此只要稍作整理，确保主键在数据库之间保持一致，Hive 和 SparkSQL 几乎可以马上使用从这些数据源采集的数据。在我们的示例中，Sqoop 是提取存储在关系数据库中的客户评价数据的理想工具。

另一方面，Flume 是用于采集日志记录的工具，但也可以从任何 HTTP 源中采集。Sqoop 用于结构化数据，而 Flume 主要用于非结构化数据，例如包含网络流量数据的日志。日志记录一般被认为是半结构化的，因为它们是需要解析的文本，但通常每一行都拥有相同的标准格式。Flume 还可以从 Web 请求中采集 HTML、XML、CSV 或 JSON 数据，因此能行之有效地处理特定的半结构化数据和非结构化数据的包装数据，如评论、评价或其他文本数据。因为 Flume 比 Sqoop 更通用，因此它不一定与下游数据仓储产品保持一致，并且一般要求采集过程和分析之间有 ETL 机制。

本书没有讨论消息队列服务。例如，Kafka 是一种分布式队列系统，可在现实世界、数据系统中的应用程序、数据湖泊三者之间创建数据边界。请求数据可以放在 Kafka 队列中，然后按需采集，而无须用户向应用程序发送请求数据，然后采集到 Hadoop 中。消息队列使得数据采集过程变得更加实时，或者至少变成了一次处理一小片，而不是像 Sqoop 那样进行大批量作业。

然而，为了获取实时数据源，还需要其他工具来处理流式数据。流式数据是指在线不断进入系统的无界的、可能无序的数据。Twitter 的 Storm（现在的 Heron）、MillWheel 和 Timely 等工具支持分布式的、容错的实时数据集处理。这些工具可以在 YARN 上运行，并在处理结束时将 HDFS 作为存储工具。与之类似，Spark Streaming 提供了流式数据集的微批次分析，允许以固定的间隔（例如每秒）将记录收集成一个批次，并一次性分析或使用它们。

许多现代分析架构将这些采集和处理工具组合，使其同时支持批处理和流式任务，这也被称为 **lambda** 架构，如图 10-4 所示。

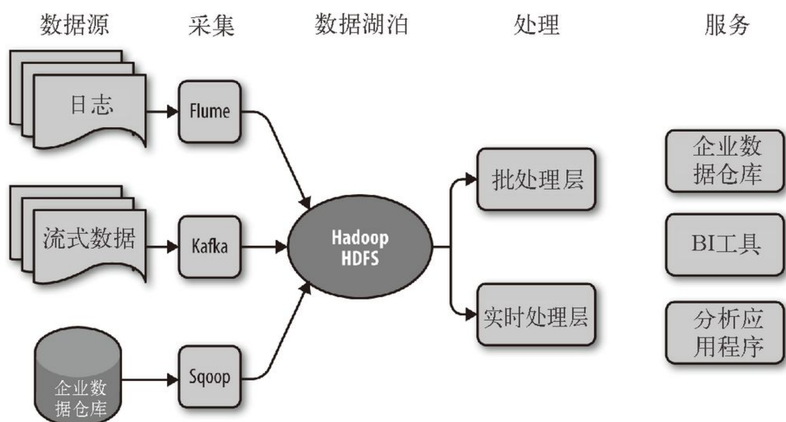


图 10-4：**lambda** 架构

当你将这些工具作为一个整体考虑时，可以清楚地看到，从直接将数据馈入数据仓储以进行分析的大规模批处理，到需要 ETL 和处理才能进行大规模分析的实时流中有一种连续性。至于具体采用哪种处理方式，主要由数据的具体速度和及时性（立即或在特定时间限制内完成分析）或完整性（近似与精确）共同决定。

10.1.3　计算数据存储

随着我们不断接近数据产品生命周期中更正式的仓储和分析阶段，分布式存储的需求被再一次唤醒。正如之前讨论的，通过将 Hadoop 作为数据湖泊来存储未处理的原始数据，我们可以获得相当灵活、敏捷的分析能力。然而，在很多使用场景中，结构和顺序也是必需的，在数据仓储中尤其如此——数据希望驻留在共享存储库中，维度模式为分析任务提供更简单、经过优化的查询。对于这些类型的应用程序，仅仅使用 HDFS 的文件系统接口与作为文件集合的数据进行交互是不够的；我们需要一个更高层次的接口，可以原生地理解 SQL 的结构化表语义。

01. 关系方法：**Hive**

在本书中，我们提出 Hive 是 Hadoop 中执行数据仓储任务的主要方法。Hive 项目包括许多组件，比如 Hive Metastore、Hive 驱动程序和执行引擎、Hive Metastore 服务和 HCatalog。其中，Hive Metastore 作为 HDFS 之上的存储管理器，存储元数据（数据库 / 表实体、列名称、类型等）；Hive 驱动程序和执行引擎将 SQL 查询编译成 MapReduce 或 Spark

作业；Hive Metastore 服务和 HCatalog 使其他 Hadoop 生态系统工具能与 Hive Metastore 进行交互。还有许多其他分布式的 SQL 或 SQL-on-Hadoop 技术来不及讨论了，它们是 Impala、Presto、Hive on Tez，等等。上述所有组件其实可以直接与 Hive Metastore 交互，或通过 HCatalog 与它交互。解决方案的选择应由数据仓储和性能要求决定，但 Hive 通常是耗时长、需要容错的查询的好选择。

在 HDFS 和 Hive 中存储数据时，一定要考虑如何以有意义且有效的方式对数据进行分区。要存储到 Hive 的话，确定分区策略时应该考虑在查询数据集时最常应用的谓词。例如，如果要分析 `WHERE year = 2015` 或 `WHERE updated > 2016-03-15` 形式的 `WHERE` 子句，很明显，按日期过滤记录将是一个重要的访问模式，因此可以按天（例如 2016-03-01）将数据分区。这让 Hive 能只读取所需的特定分区，从而减少 I/O 量并显著缩短查询时间。**1**

不幸的是，大多数 SQL 查询都非常复杂，针对分析使用的各种谓词可能会导致大量不同的分区。这要么导致数据极度碎片化，要么会降低数据存储的灵活性。除了对分布式数据执行复杂查询之外，还有第二个选择——在主要转换和过滤之后，使用 Sqoop 将数据从 Hadoop 采集出来，然后存入关系数据库中，以便能更直接地应用正常报告或 Tableau 可视化。因此，了解数据如何从许多较小的系统，流入较大的数据湖泊系统，再流回到较小的系统，是数据仓储的关键所在。

02. **NoSQL**方法：**HBase**

这里讨论的数据仓储的非关系选项是 HBase，一个列式的 NoSQL 数据库。列式数据库是 OLAP（online analytical processing，联机分析处理）类数据库访问的主力军。这类访问通常扫描大多数或所有数据库表，但只选择可用列的一部分。思考这样一个问题：每个地区每周有多少订单？这个订单表查询需要两列，分别是地区和订单日期。列式数据库只将这两列以紧凑、压缩的格式流入计算，而不采取每张表都逐行扫描（包括不需要的连接和列）的面向行的方法。因此，列式（也被称为以顶点为中心）计算为这些类型的聚合带来了巨大的性能提升。

在考虑使用哪种非关系工具和 NoSQL 数据库时，通常会有特定的要求帮你作出选择。例如，如果查询需要快速查找一个

值，则应考虑键 / 值存储；如果数据访问需求涉及稀疏数据的行级写入，并且分析主要关注聚合，那么 HBase 是很好的备选工具；如果数据是实体（顶点）之间有许多关系（边）的图，则应考虑 Titan 这样的图数据库；如果你正在使用传感器或时间序列数据，那么应该考虑 InfluxDB 这样的原生了解时间序列数据的数据库。NoSQL 数据库的数量令人惊讶，这是因为它们通常都只针对特定的使用场景进行优化。在大多数情况下，这些数据存储后端是更大、更复杂的分布式存储和计算架构的一部分。

1《Hadoop 应用架构》，Mark Grover 等人著。本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1710。——编者注

10.2 机器学习生命周期

第 5 章探索了分解数据集的抽样技术，将样本放在单个计算机上，然后使用 Scikit-Learn 来生成模型。可以序列化模型并通过分布式方法使用整个数据集对模型进行交叉验证。一般来说，这是一种非常有效的技术，被称为“最后一英里计算”。它使用 MapReduce 或 Spark 过滤、聚合或汇总数据，使数据可以加载到单个计算机的内存（例如 64GB）中，并能通过更容易获得的技术计算。此外，这也是执行没有分布式实现的计算或分析的唯一方法。

第 9 章探讨了如何使用 SparkML 库在分布式环境中执行分类、回归和聚类。过去，大数据机器学习依赖于 Mahout 库和图分析库（例如 Pregel）；而现在，SparkML 和 GraphX 库被广泛应用于分析上下文中。在一定程度上，将强大工具转换为分布式形式的趋势已经出现；但在其他情况下，分布式算法的出现比单进程版本还要早。

鉴于之前已经定义了数据产品，所以希望大家明确一点：本书中讨论的所有数据管理技术都是趋向机器学习的，以特征工程的形式为主。特征工程是分析创建决策空间的过程；也就是为了创建一个有效的模型，需要什么维度（列或字段）？其实这个过程是数据科学家的主要工作；数据产品的最终目标是如何使用前面章节讨论的工具，而不是如何设计或开发它们。

因此，了解清楚机器学习算法在期望什么，可能比直接讨论机器学习更有用。

 本书主要帮助数据科学家培养在大型数据集上进

> 行机器学习特征工程的能力。几乎所有机器学习算法都在单个实例表上运行，每一行都是学习的实例，每一列都是决策空间中的一个维度。这对在数据产品生命周期中如何选择工具有很大影响。

在关系上下文中，这意味着数据集必须在分析之前去规一化（例如将多个表连接成一个表）。这可能会给系统带来冗余数据，但这正是算法所需要的。几乎所有机器学习系统都是迭代的，这意味着系统会多次处理数据。在大数据环境中，这将导致大量开销，因此我们会使用 Spark 替代 MapReduce 来进行机器学习——Spark 将数据保存在内存中，加快每次处理的速度。

去规一化、冗余和迭代算法也对数据生命周期有影响。如果我们经常生成单个表，那么就必须问问自己，为什么最初要从数据湖泊中规一化数据。难道就不能简单地将非规一化数据直接发送到机器学习模型中吗？在实践中，Hadoop 中的模式设计高度依赖于具体的分析过程或机器学习模型的输入需求。在许多情况下，可能有多个差异很小的数据模式需求，例如所需的分区或分桶模式。虽然使用不同的物理结构存储相同的数据集通常在传统数据仓库中被认为是一种反模式，但这种方法在 Hadoop 中是有意义的——数据针对一次写入优化，存储重复数据的开销也很小。2

> 2《Hadoop 应用架构》，Mark Grover 等人著。本书已由人民邮电出版社出版，http://www.ituring.com.cn/book/1710。——编者注

考虑过机器学习构建阶段的数据存储后，第二件要思考的事是如何将模型从数据产品生命周期转移到生产中，以便将其用于识别模式、作出预测或适应用户行为！模型拟合数据，从而广泛应用于新的输入数据。拟合过程通常会产生一些模型的表示，可用于预测。例如，如果你使用朴素贝叶斯模型系列，那么拟合模型实际上是一组概率。通过这些概率中包含的实例特征的概率，我们可以计算类别的条件概率。如果你使用线性模型，那么拟合模型表示的是一组系数和一个截距，截距与自变量（特征）的线性组合产生一个因变量（目标）。

无论如何，这种表示必须从系统中导出才能运行和评估。线性模型的表示非常小，只是一组系数而已。贝叶斯模型的拟合模型可能更大一点——它是系统中每个特征和类的一组概率，因此模型表示的大小与特性数量直接相关。随机森林是多个决策树的集合，使用基于规则的方法分割决策空间。虽然每个决策树只是一个较小的树状数据结构，但是在大数据环境中，决策空间可能又大又复杂，随机森林中决策树

的数量也可能带来存储问题。模型表示越来越大，一直到 k 最近邻方法——它需要存储每个用于距离计算的训练实例来作出决策。

到目前为止，我们已经知道了导出拟合模型的两个主要机制：使用 Python 和 Scikit-Learn 序列化数据和将 Spark 模型写回 HDFS。但如果模型表示管理过程是数据产品生命周期的一部分，你会发现其他分析任务（规范化、删除重复数据、抽样等）也很有必要。

10.3　小结

大数据科学就是使用分布式计算技术进行描述性分析和推断性分析，希望这些需要分布式计算的数据的数量、多样性和速度将带来更深入或更有针对性的见解。此外，数据科学的产出是数据产品——从数据中获取价值并生成新数据的产品。因此，各种生态系统工具的集成通常围绕数据产品生命周期进行架构。

数据产品生命周期中有一个内部机器学习生命周期，它又包含两个主要阶段：构建阶段和运行阶段。构建阶段需要特征分析和数据探索；运行阶段旨在将产品的数据生成方面暴露给与数据产品交互的真实用户，生成可用于调整模型的数据，使模型更准确或更通用。通过提供数据采集、数据整理、数据探索和计算框架，数据产品生命周期提供了构建和运行模型的工作流。大多数生产架构结合了人工（由数据科学家驱动计算）分析和自动数据处理工作流，这些工作流由 Hadoop 技术生态系统提供和管理。

Hadoop 和分布式计算技术的生态系统是巨大且不断扩展的，但本书只讨论了它的基本概念，以及在评估和选择基于 Hadoop 的工具和算法实现数据产品工作流时要考虑的一些因素和作出的取舍——因为这都取决于你的需求。接下来该怎么做，是在自己的集群上进行实验和应用这些工具和模式，还是更深入地研究 Hadoop 或相关项目，都取决于你。但是希望本书介绍的概念、工具和技术为你提供了一个清晰的起点，并能持续为你的分布式数据分析之旅提供助力。

如果你读到这里了，那真的要恭喜你！你终于到达了 Hadoop 分布式数据分析指南的终点，希望这份指南是广泛且实用的。本书的目标是让你拥有足够的基础知识和背景知识，了解如何使用 Hadoop 分布式计算进行强大的大规模数据分析，并为深入了解其他一些子主题和技术作好准备。

附录 A　创建 Hadoop 伪分布式开发环境

为了执行本书中的代码，你需要设置开发环境。Hadoop 开发人员通常在伪分布式环境（也被称为单个节点设置）上测试其脚本和代码，该虚拟机在单个机器上同时运行所有 Hadoop 守护进程。

如下指导将帮助你在 Ubuntu 14.04 上使用 Hadoop 2.5.0 安装一个伪分布式环境。

A.1　快速上手

如果你不熟悉 Linux 系统管理，或者不想自己去安装 Hadoop，那么有几个选择。我们提供了一个 VMDK，供你在选中的虚拟化软件（如 VirtualBox 或 VMWare Fusion）中使用；此外，Hortonworks 和 Cloudera 都提供了可快速下载的虚拟机。

若想快速安装，只需下载虚拟机并在你最喜欢的虚拟化软件中运行它。请注意，如果你使用 Cloudera 或 Hortonworks 的发行版，那么可能与我们使用的环境略有不同。要完成所有设置，请下载预先配置好的机器或按照如下所述步骤进行。

如果你使用了我们提供的 VMDK，请使用以下用户名和密码登录机器：

```
username: student
password: password
```

如果你有信心自己设置环境，那么请继续看下一节！

A.2　设置Linux环境

在开始安装 Hadoop 之前，你需要配置一个可以使用的 Linux 环境。下面的指导假设你能在你选的机器上安装 Ubuntu 14.04 发行版——要么选择双引导配置，要么选择虚拟机。你可以凭喜好选择使用 Ubuntu 服务器版还是 Ubuntu 桌面版，但是不管怎样都需要熟悉如何使用命令行。

我们的基础环境是 Ubuntu x64 Desktop 14.04 LTS。

通过运行以下命令，确保你的系统是最新的：

```
$ sudo apt-get update && sudo apt-get upgrade
$ sudo apt-get install build-essential ssh lzop git rsync curl
$ sudo apt-get install python-dev python-setuptools
$ sudo apt-get install libcurl4-openssl-dev
$ sudo easy_install pip
$ sudo pip install virtualenv virtualenvwrapper python-dateutil
```

A.2.1 创建Hadoop用户

为了保障 Hadoop 服务的安全，确保 Hadoop 作为 Hadoop 特定的用户和组运行。此用户将能够启动与集群中其他节点的 SSH 连接，但没有能损害运行着服务的操作系统的管理访问权限。实施 Linux 权限也有助于保障 HDFS 的安全，并且是准备安全计算集群的第一步。

本教程不适用于运行阶段；但对数据科学家而言，这些权限终究能帮你减少一些麻烦，因此在开发环境中设置权限还是有用的。而且，这还能确保 Hadoop 与其他软件应用程序是分开的，有助于机器维护的条理性。

创建 `hadoop` 用户和组，然后将 `student` 用户添加到 Hadoop 组中：

```
$ sudo addgroup hadoop
$ sudo adduser --ingroup hadoop hadoop
$ sudo usermod -a -G hadoop student
```

注销并重新登录（或重新启动计算机），输入 `groups` 命令后，就能看到你已经加入了 Hadoop 组。

A.2.2 配置SSH

SSH 是必需的，必须安装在系统上才能使用 Hadoop（并且能更好地管理虚拟环境，如果你使用的是无界面的 Ubuntu 就更是如此）。通过以下命令为 `hadoop` 用户生成 `ssh` 密钥：

```
$ sudo su hadoop
```

```
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/home/student/.ssh/id_rsa):
Created directory '/home/student/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/student/.ssh/id_rsa.
Your public key has been saved in /home/student/.ssh/id_rsa.pub.
[... snip ...]
```

对所有提示输入都按回车键才能接受默认值，并创建不需要密码进行身份验证的密钥（这是 Hadoop 必需的）。由于需要没有密码的 SSH，将 Hadoop 用户与管理用户分开是一种很好的做法；但因为这是一个开发集群，我们将选取捷径，将 student 用户作为 Hadoop 用户。

为了让 SSH 能使用密钥，使用以下命令将公钥复制到 authorized_keys 文件中：

```
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 600 ~/.ssh/authorized_keys
```

你应该可以下载这个密钥，并将其用于 SSH 连接 Ubuntu 环境。要测试 SSH 密钥，可使用以下命令：

```
$ ssh -l hadoop localhost
```

如果没有要求你输入密码就完成了，那么 Hadoop 的 SSH 就已经配置成功。

A.2.3　安装Java

Hadoop 和大多数 Hadoop 生态系统需要 Java 来运行。Hadoop 需要 Oracle Java 1.6.*x* 或更高版本，通常推荐使用特定版本的 Java。不过，Hadoop 现在维护了一份报告，列出了与 Hadoop 配合良好的各种 JDK。Ubuntu 没有在 Ubuntu 仓库中维护 Oracle JDK，因为它是专利代码，所以我们将安装 OpenJDK。有关支持的 Java 版本的更多信息，请参见 Hadoop Java 版本（http://wiki.apache.org/hadoop/HadoopJavaVersions）；有关在 Ubuntu 上安装不同版本的信息，请参见在 Ubuntu 上安装 Java（https://help.ubuntu.com/community/Java）。

```
$ sudo apt-get install openjdk-7-*
```

进行快速检查，确保安装了正确版本的 Java：

```
$ java -version
java version "1.7.0_55"
OpenJDK Runtime Environment (IcedTea 2.4.7) (7u55-2.4.7-1ubuntu1)
OpenJDK 64-Bit Server VM (build 24.51-b03, mixed mode)
```

目前，Hadoop 已在 OpenJDK 和 Oracle 的 JDK/JRE 上进行了构建和测试。

A.2.4　禁用IPv6

有报告称 Ubuntu 上运行的 Hadoop 与 IPv6 有冲突。因此，自 Hadoop 0.20 以后，Ubuntu 用户一直在其集群中禁用 IPv6。虽然目前尚不清楚最新版本的 Hadoop 是否仍然有这个漏洞，但是在单个节点或者说伪分布式环境中无须使用 IPv6，因此最好禁用它，避免任何潜在问题。

通过执行以下代码行，编辑 /etc/sysctl.conf 文件：

```
$ gksu gedit /etc/sysctl.conf
```

然后将以下内容添加到文件的末尾：

```
# 禁用ipv6
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
net.ipv6.conf.lo.disable_ipv6 = 1
```

要使更改生效，请重新启动计算机。一旦重新启动，请使用以下命令检查状态：

```
$ cat /proc/sys/net/ipv6/conf/all/disable_ipv6
```

如果输出为 0，则 IPv6 是启用的；如果为 1，说明已成功禁用 IPv6。

A.3　安装Hadoop

要获取 Hadoop，你需要从其中一个 Apache 下载镜像（http://www.apache.org/dyn/closer.cgi/hadoop/common/）下载你选择的版本。如下指导将下载编写本书时带 YARN 的 Hadoop 稳定版本——Hadoop 2.5.0。

选择镜像后，在终端窗口中键入以下命令，将 http://apache.mirror.com/hadoop-2.5.0/ 替换为你选择的、最适合你所在区域的镜像 URL：

```
$ curl -O http://apache.mirror.com/hadoop-2.5.0/hadoop-2.5.0.tar.gz
```

通过确保 `md5sum` 与镜像中的 `md5sum` 匹配来验证下载：

```
$ md5sum hadoop-2.5.0.tar.gz
5d5f0c8969075f8c0a15dc616ad36b8a hadoop-2.5.0.tar.gz
```

当然，你也可以使用任何你想用的办法下载 Hadoop——使用 `wget` 命令或浏览器都可以。

A.3.1　解压

在得到压缩的 tarball 文件之后，下一步就是将它解压。你可以使用 Archive Manager，或者按照以下说明进行操作。你必须作出一个最重要的决定：要将 Hadoop 解压到何处。

Linux 操作系统依赖于分层目录结构。在根目录下，你听说过的许多目录都有特定用途：/etc 用于存储配置文件，而 /home 用于存储用户特定的文件。大多数应用程序遍布在多个位置。例如，/bin 和 /sbin 中有对操作系统至关重要的程序；/usr/bin 和 /usr/sbin 中的程序虽不是至关重要，但是系统范围的。/usr/local 用于本地安装的程序，而 /var 用于包含缓存和日志的程序数据。你可以在这篇 Stack Exchange 文章（http://bit.ly/1Tr6QuW）中了解更多有关这些目录的信息。

将 Hadoop 移至 /opt 和 /srv 目录是不错的选择。/opt 包含非打包程序，通常为源码，很多开发人员将他们的代码放在那里用于部署。/srv 目录代表服务；Hadoop、HBase、Hive 等作为服务在机器上运

行，所以这似乎也是一个好地方。此外，它还是一个很容易访问的标准位置，因此我们把所有内容都放在这里。输入以下命令：

```
$ tar -xzf hadoop-2.5.0.tar.gz
$ sudo mv hadoop-2.5.0 /srv/
$ sudo chown -R hadoop:hadoop /srv/hadoop-2.5.0
$ sudo chmod g+w -R /srv/hadoop-2.5.0
$ sudo ln -s /srv/hadoop-2.5.0 /srv/hadoop
```

这些命令解压 Hadoop，将其移至服务目录，然后设置权限。所有 Hadoop 和集群服务将存放在服务目录。最后，创建一个我们希望使用的 Hadoop 版本的 symlink，以便将来可以轻松升级 Hadoop 发行版。

A.3.2　环境

为了确保一切正常执行，来设置一些环境变量，确保 Hadoop 在正确的上下文中执行。在命令行中输入以下命令，使用 `hadoop` 用户配置文件打开文本编辑器来更改环境变量：

```
$ gksu gedit /home/hadoop/.bashrc
```

将如下内容添加到该文件：

```
# 设置Hadoop相关的环境变量
export HADOOP_HOME=/srv/hadoop
export PATH=$PATH:$HADOOP_HOME/bin

# 设置JAVA_HOME
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
```

我们还将添加一些方便的功能到 `student` 用户环境。使用以下命令打开 `student` 用户 bash 别名文件：

```
$ gedit ~/.bash_aliases
```

将以下内容添加到该文件：

```
# 设置Hadoop相关的环境变量
export HADOOP_HOME=/srv/hadoop
```

```
export HADOOP_STREAMING=$HADOOP_HOME/share/hadoop/tools/lib/
hadoop-streaming-2.5.0.jar
export PATH=$PATH:$HADOOP_HOME/bin

# 设置JAVA_HOME
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64

# 有用的别名
alias ..="cd .."
alias ...="cd ../.."
alias hfs="hadoop fs"
alias hls="hfs -ls"
```

这些简单的别名可以节省大量的打字时间！随意添加任何你认为在开发工作中可能有用的配置。

通过运行 Hadoop 命令，检查你的环境配置是否有效：

```
$ hadoop version
Hadoop 2.5.0
Subversion http://svn.apache.org/repos/asf/hadoop/common -r 1616291
Compiled by jenkins on 2014-08-06T17:31Z
Compiled with protoc 2.5.0
From source with checksum 423dcd5a752eddd8e45ead6fd5ff9a24
This command was run using /srv/hadoop-2.5.0/share/hadoop/common/
hadoop-common-2.5.0.jar
```

如果没有出现错误，并显示类似于此处的输出，则所有内容都已正确配置。

A.3.3 Hadoop配置

设置伪分布式 Hadoop 的倒数第二步是编辑 Hadoop 环境、MapReduce site、HDFS site 和 YARN site 的配置文件。这主要指的是编辑配置文件。通过在命令行中输入以下内容来编辑 hadoop-env.sh 文件：

```
$ gedit $HADOOP_HOME/etc/hadoop/hadoop-env.sh
```

该配置最重要的部分是更改以下内容：

```
# 使用的Java实现
export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64
```

接着，编辑 core site 配置文件：

```
$ gedit $HADOOP_HOME/etc/hadoop/core-site.xml
```

使用如下内容替换 <configuration></configuration>：

```
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://localhost:9000</value>
    </property>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/var/app/hadoop/data</value>
    </property>
</configuration>
```

下一步，编辑 mapreduce site 配置——通过复制模板，打开文件进行编辑：

```
$ cp $HADOOP_HOME/etc/hadoop/mapred-site.xml.template \
      $HADOOP_HOME/etc/hadoop/mapred-site.xml
$ gedit $HADOOP_HOME/etc/hadoop/mapred-site.xml
```

使用如下内容替换 <configuration></configuration>：

```
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
</configuration>
```

现在通过编辑如下文件，来编辑 hdfs site 配置：

```
$ gedit $HADOOP_HOME/etc/hadoop/hdfs-site.xml
```

使用如下内容替换 <configuration></configuration>：

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

最后，编辑 yarn site 文件：

```
$ gedit $HADOOP_HOME/etc/hadoop/yarn-site.xml
```

按如下所示更新配置文件：

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.aux-services.mapreduce_shuffle.class</name>
        <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>
    <property>
        <name>yarn.resourcemanager.resource-tracker.address</name>
        <value>localhost:8025</value>
    </property>
    <property>
        <name>yarn.resourcemanager.scheduler.address</name>
        <value>localhost:8030</value>
    </property>
    <property>
        <name>yarn.resourcemanager.address</name>
        <value>localhost:8050</value>
    </property>
</configuration>
```

编辑完这些文件后，Hadoop 就已被全副武装，成为了一个伪分布式环境。

A.3.4 格式化NameNode

启动 Hadoop 前的最后一步是格式化 NameNode。NameNode 负责管理 HDFS。这台机器上的 NameNode 将把它的文件保存在 /var/app/hadoop/data 目录中。我们需要初始化这个目录，然后格式化

NameNode 来正确使用它：

```
$ sudo mkdir -p /var/app/hadoop/data
$ sudo chown hadoop:hadoop -R /var/app/hadoop
$ sudo su hadoop
$ hadoop namenode -format
```

你会看到页面向下滚动显示了一大堆 Java 消息。如果 `namenode` 命令成功执行（/var/app/ hadoop/data 目录下应该有目录，包括一个 dfs 目录），那么 Hadoop 就设置完成并可以使用了！

A.3.5　启动Hadoop

此时就可以启动和运行 Hadoop 守护进程了。格式化 NameNode 时，用 `sudo su hadoop` 命令切换到 `hadoop` 用户。如果还是该用户，继续执行以下命令：

```
$ $HADOOP_HOME/sbin/start-dfs.sh
$ $HADOOP_HOME/sbin/start-yarn.sh
```

守护进程会启动并发出消息，内容包括记录日志的位置和其他重要信息。如果遇到有关 SSH 密钥的问题，只需要在遇到提示时输入 `y`。可以通过 `jps` 命令查看正在运行的进程：

```
$ jps
5298 Jps
4690 ResourceManager
4541 SecondaryNameNode
4813 NodeManager
4227 NameNode
```

如果进程没有运行，就一定有哪里出错了。你可以打开浏览器，访问 http://localhost:8088 来访问 Hadoop 集群管理网站——这会打开一个页面，上面有 Hadoop 标志和一个应用列表。

最后，在 HDFS 上为 student 账户准备一个空间，用来存储数据并运行分析作业：

```
$ hadoop fs -mkdir -p /user/student
$ hadoop fs -chown student:student /user/student
```

现在可以使用 `exit` 命令退出 `hadoop` 用户的 shell。

A.3.6　重启Hadoop

如果重新启动机器，Hadoop 守护进程将停止运行，并且不会自动重新启动。如果你尝试运行 Hadoop 命令，并且收到了消息“Connection refused”，则可能是由于守护进程未运行。可以使用 `sudo` 运行 `jps` 命令来进行检查：

```
$ sudo jps
```

要在关闭后重新启动 Hadoop，运行以下命令即可：

```
$ sudo -H -u hadoop $HADOOP_HOME/sbin/start-dfs.sh
$ sudo -H -u hadoop $HADOOP_HOME/sbin/start-yarn.sh
```

这些进程就会作为 `hadoop` 用户重新启动，你又可以继续了！

附录 B　安装 Hadoop 生态系统产品

除了 Hadoop 提供的核心功能之外，本书也涵盖了构建于 Hadoop 之上的其他 Hadoop 生态系统项目。在典型设置中，这些产品通常要么安装在运行 Hadoop 和 YARN 的集群上，要么通过配置连接到 Hadoop 集群。本书假设你已经在单个节点上设置和配置了伪分布式模式的 Apache Hadoop。然而，要运行单节点 Hadoop 集群和 Hadoop 生态系统产品，还有其他几个选择，本书也将对此进行讨论。

B.1　打包的Hadoop发行版

要运行 Hadoop 单机配置，最简单的方法是安装一款由主流 Hadoop 厂商提供的虚拟化 Hadoop 发行版，比如 Cloudera 的 Quickstart VM（ttp://bit.ly/1YWtzPC）、Hortonworks Sandbox（http://

bit.ly/1YWtyLy）和 MapR 的 Hadoop 沙箱（http://bit.ly/1YWtz27）。除了一个单节点 Hadoop 集群之外，这些虚拟机还包含流行的 Apache Hadoop 生态系统项目以及专有的应用程序和工具，它们都在一个简单的完整包（turn-key bundle）里。你可以使用喜欢的虚拟化软件，如 VMWare Player（https://www.vmware.com/products/player）或 Virtualbox（https://www.virtualbox.org/wiki/Downloads）来运行这些虚拟机。

B.2　自己安装Apache Hadoop生态系统产品

如果你没有使用打包的 Hadoop 发行版，而是手动安装了 Apache Hadoop，那么你就还需要手动安装和配置本书中讨论的各种 Hadoop 生态系统项目，使它们能使用安装好的 Hadoop。

在大多数情况下，在设置好的 Hadoop 环境中安装服务（例如 Hive、HBase 等）包含如下几个步骤。

(1) 下载该服务的 tarball 发布文件。

(2) 将发布文件解压到 /srv/ 目录（其中安装了 Hadoop 服务），并从发布文件创建一个符号链接指向一个简单的名称。

(3) 配置环境变量，添加服务的路径。

(4) 配置服务，以伪分布式模式运行。

在本附录中，我们将逐步安装 Sqoop，从而使用伪分布式 Hadoop 集群。这些步骤同样适用于几乎所有本书讨论的 Hadoop 生态系统项目。

B.2.1　基本安装和配置步骤

首先从 Apache Sqoop 下载镜像（http://www.apache.org/dyn/closer.cgi/sqoop/1.4.6）下载最新的 Sqoop 稳定版本，本书创作时的版本为 1.4.6。确保你是拥有管理员（sudo）权限的用户，并获取与 Hadoop 版本（在本示例中为 Hadoop 2.5.1）兼容的 Sqoop 版本：

```
~$ wget http://apache.arvixe.com/sqoop/1.4.6/sqoop-1.4.6.bin__hadoop-2.0.4
~$ sudo mv sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz /srv/
~$ cd /srv
/srv$ sudo tar -xvf sqoop-1.4.6.bin__hadoop-2.0.4-alpha.tar.gz
```

```
/srv$ sudo chown -R hadoop:hadoop sqoop-1.4.6.bin__hadoop-2.0.4-alpha
/srv$ sudo ln -s $(pwd)/sqoop-1.4.6.bin__hadoop-2.0.4-alpha $(pwd)/sqoop
```

现在使用 `sudo su` 命令切换到 `hadoop` 用户，编辑你的 Bash 配置，添加一些能提供方便的环境变量：

```
/srv$ sudo su hadoop
$ vim ~/.bashrc
```

将以下环境变量添加到你的 bashrc 配置文件中：

```
# Sqoop别名
export SQOOP_HOME=/srv/sqoop
export PATH=$PATH:$SQOOP_HOME/bin
```

然后使用 `source` 运行配置文件，将新变量添加到当前 shell 环境中：

```
~$ $ source ~/.bashrc
```

通过从 `$SQOOP_HOME` 运行 `sqoop help` 来验证 Sqoop 是否安装成功：

```
/srv$ cd $SQOOP_HOME
/srv/sqoop$ sqoop help

15/06/04 21:57:40 INFO sqoop.Sqoop: Running Sqoop version: 1.4.6
usage: sqoop COMMAND [ARGS]

Available commands:
  codegen                 Generate code to interact with database records
  create-hive-table       Import a table definition into Hive
  eval                    Evaluate a SQL statement and display the results
  export                  Export an HDFS directory to a database table
  help                    List available commands
  import                  Import a table from a database to HDFS
  import-all-tables       Import tables from a database to HDFS
  job                     Work with saved jobs
  list-databases          List available databases on a server
  list-tables             List available tables in a database
  merge                   Merge results of incremental imports
  metastore               Run a standalone Sqoop metastore
  version                 Display version information
```

```
See 'sqoop help COMMAND' for information on a specific command.
```

如果你看到任何与 HCatalog 有关的警告，暂且可以放心地忽略它们。由代码可知，Sqoop 提供了一系列导入、导出特定的命令和工具，能与数据库或 Hadoop 数据源连接。

Sqoop 进程要么通过运行 Sqoop 命令手动执行，要么由上游系统调度或触发 Sqoop 操作来执行。但是，我们将要安装的其他产品还包括启动守护进程的命令。可以使用 `jps` 命令将这些运行中的进程列出，比如所有的 Java 进程。`jps` 命令在验证所有预期的 Hadoop 进程是否运行时非常有用；例如，如果你按照附录 A 所述启动了 Hadoop，应该能看到以下进程：

```
~$ jps
10029 NameNode
10670 NodeManager
21694 Jps
10187 DataNode
10373 SecondaryNameNode
11034 JobHistoryServer
10541 ResourceManager
```

如果没有看到这些进程，请参见附录 A 和第 2 章中有关启动和停止 Hadoop 服务的讨论。

B.2.2　Sqoop特定配置

在将 MySQL 表数据导入 HDFS 之前，需要下载 MySQL JDBC 连接器驱动程序，并将其添加到 Sqoop 的 lib 文件夹中：

```
~$ wget http://dev.mysql.com/get/Downloads/Connector-J/mysql-connector-jav
~$ tar -xvf mysql-connector-java-5.1.30.tar.gz
~$ cd mysql-connector-java-5.1.30
$ sudo cp mysql-connector-java-5.1.30-bin.jar /srv/sqoop/lib/
$ cd $SQOOP_HOME
```

这让 Sqoop 能连接到我们的 MySQL 数据库。你现在应该已经在本地开发环境中成功安装了 Sqoop 和 MySQL 服务器和客户端，并配置好了 Sqoop，可以导入和导出 MySQL。

B.2.3　Hive特定配置

Hive 的安装类似于 Sqoop。但一旦安装了 Hive，就需要将其配置为在 Hadoop 单节点集群上运行。具体来说，Hive 要求配置 Hive 仓库（将包含 Hive 的数据文件）和 metastore 数据库（将包含 Hive 的模式和表的元数据）。

01. **Hive**仓库目录

    默认情况下，Hive 数据存储在 HDFS 中，位于 /user/hive/warehouse 仓库目录。我们需要确保 HDFS 中有这个目录，而且所有 Hive 用户都可以写入。如果要更改此位置，可以通过覆盖 $HIVE_HOME/conf/hive-site.xml 中的配置来修改 `hive.metastore.warehouse.dir` 属性的值。

    至于单节点配置，假设我们将使用默认仓库目录，并在 HDFS 中创建必要的目录。首先创建一个 /tmp 目录、一个 hive 用户目录和默认仓库目录：

    ```
    $ hadoop fs -mkdir /tmp
    $ hadoop fs -mkdir -p /user/hive
    $ hadoop fs -mkdir /user/hive/warehouse
    ```

    还需要设置这些目录的权限，以便它们可以由 Hive 写入：

    ```
    $ hadoop fs -chmod g+w /tmp
    $ hadoop fs -chmod g+w /user/hive/warehouse
    ```

    此外，Hive 得能写入你配置的本地 Hadoop 临时数据目录。因此，还需要确保 `hadoop` 组有写入权限，以在该路径中创建目录：

    ```
    $ chmod g+w /var/app/hadoop/data
    ```

02. **Hive metastore**数据库

    Hive 需要一个 metastore 服务后端，供它用于存储表模式定义、分区和相关元数据。Hive metastore 服务还通过 metastore 服务 API 为客户端（包括 Hive）提供访问 metastore 信息的接口。

配置 metastore 有几种不同的方式，默认的 Hive 配置使用嵌入的 metastore Derby SQL Server，提供单进程存储，Hive 驱动程序、metastore 接口和 Derby 数据库共享相同的 JVM。这个配置便于开发和单元测试，但不支持真正的集群配置，因为任何时候都只有单个用户可以连接到 Derby 数据库。能用于生产的配置将包含 MySQL 或 PostgreSQL 等数据库。

出于本章的目的，我们将使用嵌入的 Derby 服务器作为 metastore 服务。但是建议你阅读 Apache Hive 手册，安装生产级配置的本地或远程 metastore 服务器。

默认情况下，Derby 将在启动 Hive 会话的当前工作目录下创建一个 metastore_db 子目录。但如果你更改了工作目录，Derby 将无法找到以前的 metastore，并会重新创建它。为了避免这种情况发生，需要更新 metastore 配置，对 metastore 数据库的永久位置进行配置：

```
~$ cd $HIVE_HOME/conf
/srv/hive/conf$ sudo cp hive-default.xml.template hive-site.xml
/srv/hive/conf$ vim hive-site.xml
```

查找名称为 `javax.jdo.option.ConnectionURL` 的属性，将其更新为绝对路径：

```
<property>
    <name>javax.jdo.option.ConnectionURL</name>
    <value>jdbc:derby:;databaseName=/home/hadoop/metastore_db;create
    <description>JDBC connect string for a JDBC metastore</descripti
</property>
```

更新 `ConnectionURL` databaseName 后，保存并关闭该文件。

03. 验证**Hive**正在运行

现在可以从 Hive 的安装目录启动预打包的 Hive 命令行接口（CLI），验证 Hive 配置是否正确，以及它能否在伪分布式 Hadoop 集群上运行了。

从 $HIVE_HOME 目录启动 Hive CLI：

```
~$ cd $HIVE_HOME
/srv/hive$ bin/hive
```

如果 Hive 配置正确，该命令将启动 CLI 并显示 Hive CLI 提示：

```
hive>
```

你可能会看到与 Hive metastore 配置过时有关的警告：

```
WARN conf.HiveConf: DEPRECATED: hive.metastore.ds.retry.* no longer
effect.
Use hive.hmshandler.retry.* instead
```

但是，如果你看到任何错误，请根据之前的建议检查配置，然后重试。可以使用以下命令随时退出 Hive CLI：

```
hive> exit;
```

现在可以在本地和伪分布式模式下使用 Hive 来运行 Hive 脚本了。

B.2.4 HBase特定配置

HBase 在安装后需要一些额外的配置。与 Sqoop 和 Hive 不同，它需要启动守护进程才可以与我们进行交互。

解压并安装 HBase 之后，它的目录中有一个包含 HBase 配置文件的 /conf 目录。我们将编辑配置文件 conf/hbase-site.xml，将 HBase 配置为使用 HDFS 以伪分布式模式运行，并将 ZooKeeper 文件写入本地目录。使用 `vim` 编辑 HBase 配置文件：

```
$ vim $HBASE_HOME/conf/hbase-site.xml
```

然后覆盖配置文件的以下三个配置：

```
<configuration>
    <property>
```

```
        <name>hbase.rootdir</name>
        <value>hdfs://localhost:9000/hbase</value>
    </property>
    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>
    <property>
        <name>hbase.zookeeper.property.dataDir</name>
        <value>/home/hadoop/zookeeper</value>
    </property>
</configuration>
```

通过此配置，HBase 将启动一个 HBase Master 进程、一个 ZooKeeper 服务器和一个 RegionServer 进程。默认情况下，HBase 将所有目录配置为 /tmp，这意味着除非更改配置，否则服务器重新启动时将丢失所有数据，因为大多数操作系统重启时都会清除 /tmp。通过更新 `hbase.zookeeper.property.dataDir` 属性，HBase 会将数据写入 hadoop 主目录下的可靠数据路径中。

> HBase 需要有对本地目录的写入权限才能维护 ZooKeeper 文件。因为我们将作为 `hadoop` 用户运行 HBase（或者你设置的任何启动 HDFS 和 YARN 的用户），所以请确保 `dataDir` 被配置为了 Hadoop 用户可以写入的路径（例如 /home/hadoop）。

还需要更新 HBase 环境设置的 JAVA_HOME 路径。为此，在 conf/hbase-env.sh 中取消注释并修改以下设置：

```
export JAVA_HOME=/usr/lib/jvm/java-7-oracle
```

现在，HBase 应该配置正确，能在单节点集群上以伪分布式模式运行了。

启动HBase

现在就可以启动 HBase 进程了。但在此之前，应该确保 Hadoop 正在运行：

```
/srv/hbase$ jps
4051 NodeManager
```

```
3523 DataNode
3709 SecondaryNameNode
3375 NameNode
9436 Jps
3921 ResourceManager
```

如果 HDFS 和 YARN 进程没在运行，先使用 $HADOOP_HOME/sbin 下的脚本启动它们。

现在可以启动 HBase 了！

```
/srv/hbase$ bin/start-hbase.sh
localhost: starting zookeeper, logging to /srv/hbase/bin/../logs/
hbase-hadoop-zookeeper-ubuntu.out
starting master, logging to /srv/hbase/logs/
hbase-hadoop-master-ubuntu.out
localhost: starting regionserver, logging to
/srv/hbase/bin/../logs/hbase-hadoop-regionserver-ubuntu.out
```

可以使用 `jps` 命令来验证哪些进程正在运行，该命令将展示正在运行的 Hadoop 进程、HBase 和 ZooKeeper 进程、HMaster、HQuorumPeer 以及 HRegionServer：

```
/srv/hbase$
4051 NodeManager
10225 Jps
3523 DataNode
3709 SecondaryNameNode
3375 NameNode
3921 ResourceManager
9708 HQuorumPeer
9778 HMaster
9949 HRegionServer
```

可以随时使用 stop-hbase.sh 脚本停止 HBase 和 ZooKeeper：

```
/srv/hbase$ bin/stop-hbase.sh
stopping hbase..................
HBase Shell
```

HBase 启动之后，可以使用 HBase shell 连接到正在运行的实例：

```
/srv/hbase$ bin/start-hbase.sh
```

```
/srv/hbase$ bin/hbase shell
```

你会收到提示：

```
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.98.9-hadoop2, r96878ece501b0643e879254645d7f3a40eaf101f,
Mon Dec 15 23:00:20 PST 2014

hbase(main):001:0>
```

有关 HBase shell 支持的命令的文档，请使用 `help` 获取命令列表：

```
hbase(main):001:0> help
```

还可以使用 `status` 命令检查 HBase 集群的状态：

```
hbase(main):002:0> status
1 servers, 0 dead, 3.0000 average load
```

要退出 shell，只需使用 `exit` 命令：

```
hbase(main):003:0> exit
```

你现在可以在伪分布式模式下使用 HBase 了。一定要记住，在与 HBase shell 进行交互之前，必须先启动并运行 Hadoop 进程和 HBase 进程。

B.2.5 安装Spark

在本地机器上设置和运行 Spark 非常简单，一般遵循安装其他 Hadoop 生态系统的模式。按照伪分布式 Ubuntu 机器的说明，我们已经满足了 Spark 的主要要求，即 Java 7+ 和 Python 2.6+。确保 `java` 和 `python` 程序在路径上，并且设置了 `$JAVA_HOME` 环境变量（如前所述）。

在之前的安装说明中，我们使用 `wget` 或 `curl` 直接从 Apache 镜像获取了 tarball 文件。但对于 Spark 来说，情况略有不同。打开浏览器并按照以下步骤下载正确版本的 Spark。

(1) 进入 Spark 下载页面（http://spark.apache.org/downloads.html）。

(2) 选择最新的 Spark 版本（在创作本书时为 1.5.2），并确保选择了一个针对 Hadoop 2.4 或更高版本预构建的软件包，直接下载。

Spark 的版本更新往往比较频繁。为了确保可以下载到新版本的 Spark 并立即使用它们，我们将把 Spark 软件包解压到服务目录中，然后将该版本符号链接到一个泛化的 **spark** 目录。要更新版本的话，只需下载最新版本，并将符号链接重定向到它即可。通过这种方式，新版本也将拥有所有的环境变量和配置！

首先，遵循标准惯例来安装 Hadoop 生态系统服务：

```
$ tar -xzf spark-1.5.2-bin-hadoop2.4.tgz
$ mv spark-1.5.2-bin-hadoop2.4 /srv/spark-1.5.2
```

然后，创建 Spark 的符号链接版本：

```
$ ln -s /srv/spark-1.5.2 /srv/spark
```

编辑 Bash 配置文件，将 Spark 添加到 `$PATH` 并设置 `$SPARK_HOME` 环境变量。如前所述，我们将切换到 Hadoop 用户，但你也可以将以上所有内容添加到 `student` 用户的配置文件中：

```
$ sudo su hadoop
$ vim ~/.bashrc
```

将以下内容添加到配置文件：

```
export SPARK_HOME=/srv/spark
export PATH=$SPARK_HOME/bin:$PATH
```

接下来，使用 `source` 运行配置文件（或重新启动终端），将这些新变量添加到环境中。完成之后，你应该能够运行一个本地 `pyspark` 解释器：

```
$ pyspark
```

```
Python 2.7.10 (default, Jun 23 2015, 21:58:51)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.prope
[… snip …]
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.5.2
      /_/

Using Python version 2.7.10 (default, Jun 23 2015 21:58:51)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
```

至此，Spark 已成功安装，可在本地计算机上以独立模式运行，也足以运行本书的例子了。如果你想测试 Spark/Hadoop 连接，还可以使用 `spark-submit` 将作业直接提交给以伪分布式模式运行的 YARN 资源管理器。有关此主题和其他主题的更多信息，比如在 EC2 上使用 Spark 或使用 iPython notebook 设置 Spark，请参阅 Benjamin Bengfort 的“Getting Started with Spark (in Python)”（http://districtdatalabs.silvrback.com/getting-started-with-spark-in-python）。

最大限度简化Spark

Spark（和 PySpark）的执行可能非常冗长，会有许多 INFO 日志消息打印到屏幕上。这在开发过程中特别烦人，因为 Python 堆跟踪或 `print` 语句的输出可能会丢失。为了降低 Spark 的冗长度，可以在 $SPARK_HOME/conf 中配置 log4j 设置，如下所示：

```
$ cp $SPARK_HOME/conf/log4j.properties.template \
      $SPARK_HOME/conf/log4j.properties
$ vim $SPARK_HOME/conf/log4j.properties
```

编辑 log4j.properties 文件，将代码中的每一行的 `INFO` 替换成 `WARN`，类似于：

```
# 设置：让所有事件记录到控制台
log4j.rootCategory=WARN, console
log4j.appender.console=org.apache.log4j.ConsoleAppender
log4j.appender.console.target=System.err
log4j.appender.console.layout=org.apache.log4j.PatternLayout
log4j.appender.console.layout.ConversionPattern=%d{yy/MM/dd HH:mm:ss} %p %
```

```
%m%n

# 减少输出第三方冗杂的日志
log4j.logger.org.eclipse.jetty=WARN
log4j.logger.org.eclipse.jetty.util.component.AbstractLifeCycle=ERROR
log4j.logger.org.apache.spark.repl.SparkIMain$exprTyper=WARN
log4j.logger.org.apache.spark.repl.SparkILoop$SparkILoopInterpreter=WARN
```

再次运行 PySpark，输出消息应该简洁多了！

术语表

可访问的

在一个计算集群上下文中，如果一个节点可以通过网络到达，那么它就是可访问的；在其他上下文中，如果一个工具或者库能轻易为特定人群使用和理解，那么它就是可访问的。

累加器

一个共享变量，只能应用满足结合律的运算，如加法（特定于 Spark，在 MapReduce 中称为计数器）。因为满足结合律的运算是与顺序无关的，所以无论运算顺序如何，累加器都可以在分布式环境中保持一致。

动作和转换

请参见“转换和动作”。

代理

代表用户例行运行的服务，通常是后台进程，独立执行任务。Flume 代理是构建数据流的基本单元，它从源中采集和整理数据，最终通过通道将数据传输到数据槽。

匿名函数

没有指定识别符（变量名称）的函数。这些函数通常在运行时构造，并作为参数传递给高阶函数；也可以用它们轻松创建闭包。传递

匿名函数给 Spark 操作来定义它们的行为。另请参见“闭包”和“lambda 函数”。

应用程序编程接口（**application programming interface，API**）

用于指定软件组件如何交互的例程、协议和接口的集合。MapReduce API 指定用于构建 `Mapper`、`Reducer` 和 `Job` 子类的接口，定义 MapReduce 行为。与之类似，Spark 也有可以应用于 RDD 的转换和动作的 API。

ApplicationMaster

在 YARN 中，ApplicationMaster 是特定于框架的库（例如本书中的 MapReduce、Spark 或 Hive）的实例。ApplicationMaster 从 ResourceManager 协商取得资源，在 NodeManager 上执行进程，跟踪作业状态并监视进度。

满足结合律的

在数学中，不管满足结合律的运算如何分组，只要顺序保持不变，计算结果便相同。满足结合律的运算在分布式环境中很重要，因为它让你能在计算最终整体之前让多个处理器同时计算分组子操作。

Avro

Apache Avro 是 Apache Hadoop 内开发的一个远程过程调用（remote procedure call，RPC）数据序列化框架，它使用 JSON 定义模式和类型，然后以紧凑的二进制格式对数据进行序列化。

bag of words

在文本处理中，bag of words 是一个模型，根据最重要的令牌或单词的频率或出现编码文档，不考虑令牌的顺序。

偏差

在机器学习中，因偏差引起的误差是模型的预期平均预测与正确值之间的差异。偏差通常用来衡量模型的错误程度。随着偏差的增大，方差逐渐减小。另请参见“方差”。

大数据

利用极大数据集探寻与人类行为和交互特别相关的模式、趋势和关系的计算方法学。大数据具体指的是海量、难处理和稍纵即逝的数据，单台机器无法进行可靠计算。因此，大数据技术主要利用分布式计算和数据库技术来计算结果。

二元短语（**bigram**）

字符串或数组中的两个连续令牌的序列。令牌通常是字母、音节或单词。二元短语是 n-grams 的特定形式，其中 $n = 2$。

块

块是 HDFS 存储大文件的方法，将大文件分割成相同大小（通常为 128MB）的数据块。块在多个 DataNode 上都有副本（默认复制因子为 3），通过冗余提供数据持久性，并支持数据本地计算。

布隆过滤器

一个紧凑的概率数据结构，可用于测试某些数据是否是集合的成员。可能存在误算率（以为元素是集合的成员，而实际上不是），但可以通过调整过滤器的大小来设置误算出现的概率。漏报（以为元素不是集合的成员，而实际上是的）不会出现，因此布隆过滤器的召回率高达 100%。

广播变量

广播变量在 Spark 中是一种机制，用于创建按需传输到集群中每个节点的只读数据结构。广播变量可以包含计算所需的额外信息、先前转换的结果或查找表。因为它们是只读的，所以是集群安全的。另请参见“分布式缓存”。

构建阶段

在机器学习中，构建阶段通常通过一些迭代优化过程将模型形式拟合到现有数据。构建阶段包括特征提取、特征变换，以及正则化或超参数调整。构建阶段的输出是拟合模型，可用于进行预测。

字节数组

由固定长度的单字节数组构成的数据结构，可以存储任何类型的信息（数字、字符串、文件的内容），并且非常一般化，因此作为行键用于 HBase。另请参见“行键”。

cascading

Driven, Inc. 的不关心规模的数据应用程序开发框架，为 MapReduce 提供了高级抽象。通常用于将数据流或多部分作业定义为有向无环图。

集中管理集群

一个拥有角色不同的两个节点的计算集群，分别是 manager（master 节点）和 worker 节点。集群中有一个或多个协调管理节点，集中决策，无须共识。管理节点负责数据的完整性、协调性、一致性，并处理客户端请求。另请参见"对等集群"。

质心

在无监督机器学习（聚类）中，质心是特征空间中的一个点，定义了集群的中心。尽管不是所有聚类算法都有质心（生成中心），但是那些有质心的算法将质心定义为簇中所有点的距离的平均。

通道

在计算机科学中，通道指信息流动的途径。这里指的是 Apache Flume 中的通道，它们是被动存储区或缓冲区，保存事件信息直到它们被下游数据槽收集。

点击率

衡量以引导用户浏览额外信息为目的的电子邮件、网页或广告的有效性。当用户单击超链接时，处理超链接请求的服务器会写入一条日志记录，从而获取点击率。例如，可以测量购物车应用程序中"购买"按钮的点击率。

client

一般来说指某个计算服务或资源的请求者，通常是人类用户。本书提到的 client 指发送 Web 请求、提交 MapReduce 作业的人，或 Spark 应用程序中驱动程序所在的计算机。也可以由代理代表 client 进行日常工作。

闭包

一个绑定到自己的封闭执行环境的函数，环境中有约束变量。因

为环境将被赋予的变量映射到外部函数，所以这些变量不能被外部进程修改，这使得闭包可用于分布式上下文。

云计算

使用远程数据中心的共享计算资源（而不是利用本地服务器或个人设备）。共享计算资源通常是弹性的，这意味着你可以根据需要扩大和收缩资源的使用和分配。

集群

通常指执行集体或相关计算的设备的集合。在 Hadoop 中，指运行 HDFS 和 YARN 守护程序的一组服务器或计算机。

系数

在线性模型中，系数是定义因变量空间中的超平面的数值向量。你可以通过求变量向量和系数的点积（或线性组合）来预测目标值。

协同过滤

基于许多用户（协同）的集体偏好，自动提供推荐（过滤掉大量可能的项目）的方法。协同过滤技术通常是使用机器学习算法开发的模型，用于作出影响用户行为的预测。

collector

一种特殊的 Flume 代理类型，用于监听来自多个上游代理的数据，聚合其输出，然后将输出收集到日志文件、HDFS 或 HBase。

列族

HBase 中的一组相关列，共享相同的前缀，并且逻辑上和物理上都存储在一起。

面向列 / 列式数据库

内部按列（而不是按行）存储数据的数据库系统，尤其适合于在选定列执行聚合的 OLAP 用例。

满足交换律的

在数学中，无论运算以什么顺序执行，满足交换律的运算都返回

相同的结果。满足交换律的运算在分布式计算中很重要，因为它允许数据以任何顺序进入，并返回相同的结果。

可比较的

具体是指可以使用不等式（例如大于）来比较两个对象。Java 和 Python 都提供了数据模型，通过定义必须返回不等式结果的方法，允许对象比较。

复杂键

不是简单或原始类型（例如整数或字符串）的键（例如键 / 值对中的键）。大多数复杂键是复合键的形式；其他类型的复杂键可以是嵌套数据类型（例如字典），或可以表示任意数据的字节数组。另请参见“复合键”。

复合键

由两个或多个简单键组成的键（例如键 / 值对中的键），通常存储在元组中。复合键对于 map 阶段和 reduce 阶段之间数据组织，以及作业之间的数据组织很重要。

计算

在本书中，指使用计算机处理器对某些数据执行计算。这里的计算与“存储”不同，它作用于数据输入并产生数据输出。

无冲突的复制数据类型（**conflict-free replicated data type，CRDT**）

一种特殊的数据结构，可以通过满足结合律和满足交换律的运算并发地变化。CRDT 在分布式系统中提供最终的一致性和单调性（不能回滚）。另请参见“累加器”和“计数器”。

一致性

分布式计算的属性，其中单个任务的失败不会影响最终结果；也表示分布式系统中的所有节点都会看到相同的数据视图。

列联表

具有两个维度的表格，允许检查分类变量之间的关系。表维度的

交点（每个单元格）包含每个维度的分类值的共现频率。

计数器

在 MapReduce 中，计数器是一个共享变量，只能以固定的量递增。因为求和是满足结合律的，递增的顺序并不重要，所以可以在分布式环境中放心使用计数器。另请参见“累加器”。

守护进程

一种在后台运行的计算机软件，不需要用户输入。通常作为一种服务，监听网络上传入的信息并进行适当响应（服务器）。

数据分析师

主要关注数据产品开发的描述和推测的数据科学家，工作内容通常与建模、特征工程和探索相关。另请参见“数据建模师”。

数据应用程序

一种软件应用程序，用于处理大量的领域特定数据。例如，Microsoft Excel 是一种数据应用程序，用于处理电子表格或财务方面的数据。另请参见“数据产品”。

数据工程师

主要关注数据技术的数据科学家，工作通常与软件开发、数据库工具和计算基础设施有关。

数据流

在数据流中，一个单位的数据或事件（例如单个日志语句）从源流经一系列跃点到达下一个目的地。

数据湖泊

一个存储系统，用于以原始（被采集的）格式存储大量原始数据，一般采用无格式或半结构化格式。通常在数据湖泊中应用提取、转换和加载（ETL）操作来提取本地数据集市，将其用于下游计算。

数据本地计算

一种分布式计算概念，旨在减少所需的网络流量。节点对其本地

存储的数据进行计算，而不试图从集群中的其他位置获取数据。

数据挖掘

分析不同来源的数据以生成新信息或获取更深入洞见的过程。

数据建模师

数据科学家的分支，根据统计和机器学习模型对数据进行探索和解释。

数据并行

一种在多个处理器之间进行计算的方法，其中数据分布在不同的节点上，各个节点同时对数据应用相同或相似的计算。

数据产品

自适应的、广泛适用的经济引擎，从数据中获取价值，通过影响人类行为或通过对新数据进行推论或预测，产生更多数据。

数据科学

创建和开发数据产品所涉及的工作流和过程。

数据科学流水线

描述数据科学分析过程的教学模式。流水线规定了一个线性过程，数据在其间被采集、整理、计算、建模并最终得以可视化。

数据科学家

他们是拥有强大统计学背景的程序员，是具备高超程序设计能力的分析师，是非常了解数据如何影响可视化的设计师，或是在构建数据产品方面富有创新思想的领域专家。在任何情况下，数据科学家都是全能的通才，能够轻松学习新的方法来处理数据。

数据仓库

大型数据存储，通常为关系型，包含一个组织多个维度或多个方面的数据。数据仓库通常以“星型模式”组织，以便在事务成本和在线异步处理之间取得平衡。另请参见“企业数据仓库”（enterprise data warehouse，EDW）。

数据库

简单来说，就是以电子格式存储的数据的集合。然而，它通常是“数据库管理系统”的缩写。数据库管理系统是一个软件应用程序，负责存储在磁盘上的数据的组织、管理和访问。

DataFrame

一种数据结构，指的是以行（案例或实例）和列（特征或度量）结构化的表格数据。DataFrame 从 R 编程语言就开始流行，并在 Python（通过 Pandas 库）和 SparkSQL（现在的 Spark DataFrame）中实现。

DataNode

指 HDFS 中，运行在集群中每个存储节点上的服务，提供数据复制。DataNode 与 NameNode 连接，以提供和分布式存储状态有关的信息，并响应客户端对文件系统操作的请求。

决策空间

指机器学习中，由实例特征给定的多个维度定义的空间中的一个区域，决策是针对实例特征的。决策空间越大，模型的通用性越强。另请参见“特征空间”。

声明式语言

指编程中的一种非命令式语言，让程序员在不明确列出从输入到输出应采取什么步骤的情况下描述所需结果。SQL 是一种声明式语言，而 Python 不是。

去规范化

通过分离关注点，规范化的数据存储在多张表中。去规范化指将这些数据描述为单张表的过程，通常通过使用 `JOIN` 函数。去规范化的数据以冗余为代价，形成集中的单个完整的记录。

反序列化

将数据（通常是软件中的对象）的字符串或字节表示加载或转换回可由程序使用的运行表示的过程。

分布式缓存

一种 MapReduce 工具，类似于 Spark 的广播变量。在执行任务之前，用于计算的所有节点所需的文件（例如停用词列表、查找表等）从 HDFS 被复制到每个 worker 节点。

分布式计算

一种软件或计算系统，其处理组件位于多台计算机上，互相之间通过网络通信，通过消息传递协调计算。分布式计算允许多台计算机并行工作，提供了性能优势。但是由于协调需求，它通常要求算法的结构针对分发专门设计。

分布式存储

数据存储在安装在多台主机上的多个磁盘上。为了访问数据，需要网络流量来定位和获取所请求的数据。分布式存储确保了数据本地计算会发生，因为数据已经位于将发生处理的地方了。此外，大多数分布式存储系统还复制数据，让多个主机存储数据的冗余副本，防止数据丢失。

领域专家

数据团队成员，对建模领域有深入了解。特征工程过程需要领域专家在预测性和系统方面的直觉来指导模型运行。领域专家通常也是敏捷开发的客户，为工程和分析过程提供指导。

企业数据仓库（**EDW**）

用于支持企业级数据报告和分析的中央数据存储库，被视为大多数商业智能环境的核心组件。

可执行文件

可以在计算机上执行的程序。Hadoop Streaming 可将可通过 `$PATH` 变量定位的可执行程序作为 mapper 或 reducer。例如，Python 可执行文件（拥有执行权限和指定解释器的 shebang 的 Python 脚本）可用于 MapReduce 编程。

执行计划

图或树，用于描述可执行过程或函数的顺序和数据流。Spark 应

用程序通过一系列转换和一个最终被应用的动作来定义执行计划。SQL 和 HiveQL 是声明式语言，必须由底层系统转换为执行计划。

executor

在 Spark 中，executor 是运行在每个 worker 节点上的一个进程，代表集群管理器（YARN 上的 Spark ApplicationMaster）和驱动程序中的 SparkContext 来管理任务和数据服务。

容错性

如果一个组件出现故障，不应该导致整个系统出现故障。系统应优雅地降级到较低的性能状态。如果故障组件恢复，应该能重新加入系统。

特征空间

指机器学习中，由实例的属性（也被称为特征）定义的空间。特征空间还包括向更高维度的映射——对特征应用函数从而创建新值。例如，给定有 6 个因变量的一般线性模型，在特征空间中有 6 个维度来适应超平面。但对于二次多项式回归，由于平方映射将应用于原先的 6 个因变量，所以特征空间将有 12 个维度。另请参见“决策空间”。

过滤

在函数式编程中，过滤器是一个函数。它接受另一个函数和一个集合，并返回一个新的集合，新集合仅包含映射到过滤函数返回 `True` 的元素。换句话说，过滤函数是一个测试，确定一个元素是否属于一个较小的新集合。

第一范式

规范化数据库中的关系（表）的属性，使得每列只包含原子值，而每行的该列只包含一个值。例如，在这种范式下，属性不能是列表。

拟合模型

将超参数化模型形式拟合到数据的结果，通常通过优化函数，使模型参数能够基于新数据进行预测。拟合模型是机器学习训练的产物。

函数式编程

一种编程风格，计算在其中被视为函数的评估，用于避免状态改变或数据突变。因此，函数式编程是分布式计算的理想选择，因为需要固定状态和函数处理来确保协调一致。

可通用的

在机器学习中，如果一个模型可以根据未知的输入数据作出很好的预测，那么就称它为可通用的。如果一个模型拟合不足，那么就不能将该模型推广到更大的决策空间；如果一个模型过度拟合并仅是记住了数据，那么即使在本地决策空间中，它对未知数据的预测也是不正确的。

产生式模型

使用联合概率分布，而不是条件概率分布（判别式模型）来确定数据如何生成的模型。针对分类器，生成式模型对什么分类最可能产生信号作出了解答。

全局解释器锁（**global interpreter lock**，**GIL**）

解释型语言中的一种机制，用于同步线程的执行，确保任何时候只有一个线程在执行，以保护非线程安全的内存。GIL 是 Python 结构上的一部分，让 Python 先天不支持并发性；若想在 Python 中实现并行性，必须使用多个进程，且每个进程都有自己的 GIL。

Google 的 **BigTable** 架构

BigTable 是一种分布式存储系统，用于管理可以扩展到非常大的结构化数据。Chang 等人在 Google 2006 年的论文“Bigtable：A Distributed Storage System for Structured Data”中对此进行了详细讨论。

图形分析

一种分析，用于评估结构为以边相连的顶点的数据。顶点和边都可以包含数据，并且可以通过遍历来计算这种形式的数据集（而不是矩阵形式的数据集）。遍历本质上是可并行化的，因此，图形算法可以立即应用于大数据环境中。Spark GraphX 这样的库提供了图形分析工具。

Hadoop Pipes

一个内部 MapReduce 系统，允许 C＋＋ 代码访问 HDFS 并执行 mapper 和 reducer。Pipes 与 Streaming 类似，都将 Pipes 代码分割为独立的应用程序特定的库。但与 Streaming 不同的是，Pipes 支持类型化的字节序列化和更全面的 API。

Hadoop Streaming

MapReduce 应用程序的实用程序，允许将任何可执行文件用作 mapper 和 reducer。Hadoop Streaming 本身就是一个 MapReduce 应用程序，它通过标准输入将数据传输到可执行文件，然后通过标准输出和标准错误从可执行文件中收集信息。Hadoop Streaming 让 Python 和 R 开发人员能编写 MapReduce 代码。

可散列的

在 Python 中，如果一个对象拥有一个在其生命周期内永远不会改变的散列值，那么该对象就是可散列的。因此，可散列的对象是不可变对象，或者是通过其内存地址进行散列的类的实例。所有可以用作字典中的键的内容都是可散列的（例如非列表或其他字典）。

HDFS

Hadoop 分布式文件系统，Hadoop 的两个主要组件之一。HDFS 通过在集群上实现三种类型的服务来提供分布式存储，分别是 NameNode、Secondary NameNode 和 DataNode。

高基数

指列或数据属性的值非常罕见或各不相同（每条记录都有一个值）。高基数的列难以分析或聚合，自动数据类型检测也通常不起作用。

Hive

一种为 HDFS 中存储的数据提供类 SQL 接口的系统。Hive 让数据科学家能将 Hadoop 视为分布式数据仓库，并能以结构化方式并行执行 OLAP 操作。

Hive CLI

Hive 命令行接口，与 Hue 打包在一起。它提供了一个交互式 shell，用于使用 Hive 并执行 HiveQL 语句。

Hive metastore

Hive 使用的数据库，用于存储 HDFS 上有关 Hive 表和分区的元信息。

HiveQL

Hive 查询语言，属于 ANSI SQL 子集的 Hive 数据定义语言（Data Definition Language，DDL）。

假设驱动开发

敏捷数据产品开发方法，用假设替代需求，并尝试使迭代敏捷开发过程与实验、观察和重新定义的迭代科学方法模型相匹配。

恒等函数

一个总是返回与参数相等的值的函数。在数学中，这表示为 $f(x) = x$。

不可变的

不随时间改变或无法改变。在 Python 中，不可变对象在运行时无法修改，例如元组、字符串、整数或布尔值。不可变对象提供了许多好属性，例如安全性（对象被传递给函数时不会发生意外突变）、紧凑性（所需内存减少）和散列的可比性。

索引

从较长形式的数据导出汇总数据结构的计算，以便快速查找各个记录。索引作为一个预处理步骤，用来提高下游计算的速度。

采集

数据采集是指从外部源收集数据并在本地计算环境管理数据的手动或自动过程。在大数据环境中，采集通常意味着并行处理输入数据流，以便及时获取数据。

输入 / 输出

在编程中，输入是提供给进程或函数用以计算的数据，计算的结果就是输出。通常，此形式中的输入 / 输出（input/output，I/O）是指从磁盘收集数据，将其发送到处理器，最后将结果写回磁盘的过程。

交互式分析

一种技术，将计算机在大量数据上进行重复任务的计算能力，与能够从全局层面识别模式和一般性的人类认知理解力相结合。交互式分析可以引导自动模型生成，或通过可视化来调整模型的行为。

倒排索引

一种特殊索引，将单词、数字、用户或重要信息等内容映射到它们在数据库、文件、文档或文档集合中的位置。

迭代计算

一种重复计算，其中单个计算块被定义为迭代；每次迭代重复，使上一个迭代的输出成为下一个迭代的输入。迭代和递归是计算机算法的基本构件。

迭代数据处理

迭代计算的一种，指一种算法对相同数据进行多次处理，将每个迭代的结果传递到下一次迭代，但不改变数据。优化就是迭代数据处理的一个例子，一次数据处理用于计算误差，下一次迭代修改参数以缩小误差，之后算法继续迭代，直到误差低于某个小阈值。

Java 数据库连接（Java database connectivity，JDBC）

基于 Java 的接口，允许客户端使用兼容的适配器访问支持 JDBC 的数据库。Sqoop 使用 JDBC 连接器与第三方数据库集成。

作业

在分布式计算中，作业是指完整的计算，由多个并行运行的独立任务组成。

作业链

MapReduce 应用程序中使用的一种技术，通过将前一个作业的

输出用作下一个作业的输入，将一个或多个 MapReduce 作业链接在一起，从而构建更复杂的算法。

作业客户端

客户端是作业的发起者，是最关心结果的一方。客户端可以在作业运行期间保持连接，也可以先让作业在集群上独立运行，过后再返回以查找结果。

作业配置

用于定义范围的作业参数，例如应该使用的 mapper、reducer 或 executor 的数量。

Jupyter notebook

以前的 iPython notebook。notebook 是结合了可执行代码和富文本的文档，旨在以一种演示文稿的格式展示分析及其结果。因此，它们被广泛用于分析，从而显示可重现的结果。

Kerberos

用于验证服务请求的安全方法，可用于 HDFS 和 YARN API 以及集群保护。

键／值

一种联结的数据项，其中键是与数据值相关联的唯一标识符。键／值对将关系（由键定义）分发到多个处理器，然后聚合（reduce）其结果。

键空间

系统中用于计算的键／值对中键的域。键空间定义了数据如何分区到 reducer，以及键值对如何分组和比较。

lambda 架构

一种系统设计，能应对不断采集到的、需要使用分布式计算框架（如 MapReduce 或 Spark Streaming）及时处理的大量数据。lambda 架构使用消息队列前沿来缓冲传入到处理速度可能较慢的应用程序中的数据。这些应用程序执行初步计算并将结果存储在 speed

表中，执行最终计算并将结果存储在 batch 表中。客户端查询近似的 speed 表及时获取结果，依靠 batch 表进行更准确的分析。

lambda 函数

在 Python 中，lambda 关键字用于定义未绑定到标识符的匿名函数。另请参见“匿名函数”和“闭包”。

延迟执行

一种将表达式评估延迟到必要时刻的策略，从而达到最小化计算和重复性的目的。在 Spark 中，应用于 RDD 的转换操作通过生成转换过程（lineage）图来延迟执行，仅在对 RDD 应用动作操作时才执行。

词汇多样性

自然语言语料库中的单词数量与词汇量的比例，例如一个单词在语料库中的平均使用次数。词汇多样性用于监测文本数据的异常变化。

转换过程（**lineage**）

在 Spark 中，每个 RDD 存储通过应用转换从其他数据集构建出它的机制。转换过程让 RDD 能在故障时本地重建，并为 Spark 中的容错性提供基本机制。

线性作业链

一种作业序列，前一个作业的输出用作后一个作业的输入。另请参见“作业链”。

log4j

一个开源 Java 项目，允许开发人员控制日志消息输出的粒度。在 Spark 或 MapReduce 中修改 log4j 设置可以最小化控制台输出的数量，使分析人员能更轻松地了解结果。

机器学习

发现数据模式，利用这些模式构建模型，对新数据进行预测或估计的技术。

map

一种函数式编程技术，其中函数被应用于集合中的每个单独元素，生成一个新集合作为每个 map 的输出。map 本质上是可并行的，因为将 map 函数应用于元素不依赖于任何其他 map。

master 节点

集群中的一个节点，实现了一个主守护进程（用于管理集群中的存储和计算的进程）。主进程包括 ResourceManager、NameNode 和 Secondary NameNode。

最大值

在描述性统计中，数据集中的最大值。

平均值

在描述性统计中，通过将数据集中数值之和除以值的数量，来描述数据的集中趋势的值。

中位数

在描述性统计中，有序数据列表中的中间值。

微框架

一个术语，指简约的应用程序框架。本书使用 Python 和 Hadoop Streaming 构建了一个 MapReduce 的微框架。

最小值

在描述性统计中，数据集中的最小值。

众数

在描述性统计中，数据集中最常出现的值。

模型族（**model family**）

在机器学习中，模型族从宏观角度描述了决定预测的相关变量之间的联系。例如，基于系数向量与因变量向量的线性组合，线性模型描述了对连续目标值 Y 的预测。

模型形式（**model form**）

在模型拟合之前对模型轮廓的说明，特别定义了超参数，以及拟合模型的特征空间。例如，给定支持向量机模型族，模型形式可能是拥有 RBF 核函数、伽马值为 0.001 且松弛变量为 1 的 SVM。

munging

munging 最初来自麻省理工学院模型训练小组，指的是将数据混合在一起形成的统一或规一化整体，可能具有破坏性。

NameNode

HDFS 的 master 节点，负责集群 DataNode 的集中协调。NameNode 分配存储资源，并将大文件拆分成块，在集群中复制。NameNode 还将客户端直接连接到要访问数据的 DataNode。

节点

集群中的单个机器，能实现一些服务，特别是守护进程服务（如 NodeManager 和 DataNode）。

NodeManager

YARN 中在集群中的每个节点上运行的进程或代理。NodeManager 负责跟踪和监视各个 executor（容器）的 CPU 和内存的使用情况以及节点的运行状况，并将其报告回 ResourceManager。NodeManager 还通过调度 executor（容器）在本地进行工作，来代表 ApplicationMaster 执行框架作业。

NoSQL

指“不仅仅是 SQL”（not only SQL）或者“非关系型”（not relational）。NoSQL 最初是一个标签，在一场讨论数据库技术（如 Cassandra、HBase 和 MongoDB）的会议中使用。NoSQL 现在指不符合传统关系数据库管理系统定义的数据库，它们提供的领域特定数据模型（如图或列）通常是分布式的。

大数据操作系统

Hadoop 的两大支柱服务——HDFS 提供分布式数据存储、YARN 提供集群计算资源管理——为集群计算提供了平台，也使它成为了大

数据的操作系统。

运行阶段

在机器学习中，构建阶段之后就是运行阶段。拟合模型在此阶段进行预测（回归分析作出连续值估计、分类器分配类别、聚类确定归属）。

上线

在数据产品中使用拟合模型。另请参见“运行阶段”。

pairs 和 stripes

在矩阵（例如单词共现矩阵）上执行分布式计算的两种方法。在 pairs 方法中，矩阵中的行 i 和列 j 的每个单元格单独映射，如 (i, j)/值；在 stripes 方法中，每行 i 作为完整的值被映射，通常作为列 j 的关联数组。

Pandas

一个开源库，提供易于使用的数据结构，如 Series 和 DataFrame，可以应用多个数据分析工具。

并行

两个计算同时运行。

可并行的

如果可以将一个算法分解成可以同时运行的离散任务，则该算法被称为可并行的。可并行的算法有一个属性：可并行运行的任务越多，算法的完成速度就越快。

并行化

算法向可并行形式转化的过程。

对等集群

与集中管理的集群相反，对等集群是完全去中心化的，没有控制源。执行对等协调的算法不能依赖于一个集中的控制中心。Hadoop 和 Spark 是集中管理的集群，而 Bitcoin 这样的应用程序是完全去中

心化的，被称为对等分布式计算。

Pig

Pig 是一个大数据框架，由 Pig Latin（一种用于表达数据分析程序的高级语言）和一个编译器组成，后者可以将 Pig Latin 翻译成在 Hadoop 上执行的 MapReduce 作业序列。

POSIX

“可移植操作系统接口”（portable operating system interface）是 IEEE 计算机学会创建的一系列标准，旨在提高操作系统之间的兼容性。

预测模型

一种统计工具，通过推断技术来描述未来可能发生的行为。

过程化语言

与声明式语言相反，过程化语言定义了一个必须逐个执行的有序命令集。Python 可以使用过程化风格来编写，当然用函数式风格或面向对象风格也没问题。

进程

进程是正在执行的计算机程序的实例，包括完整的计算环境和资源。进程可以由并发运行的多个执行线程组成；但是一般来说，当在分布式环境中讨论进程时，是指一个必须通过网络与其他程序通信的独立程序。

产品印象

在线营销术语，指单一用户有机会查看特定的产品，通常是与超链接相关联的产品。通过数据采集技术，我们可以通过比较生成产品印象的网络日志及相关的点击率来监控产品印象是否成功。

投影

投影是一种操作，针对由一组属性定义的关系（表）。投影输出一个新关系，丢弃或排除原始关系中不在投影里的属性；换句话说，投影会移除表中的列。

PySpark

交互式 Python Spark shell，被实现为命令行 REPL（read, evaluate, print loop，读取、评估、打印循环），并由 `pyspark` 命令启动。

Python Spark 应用程序

由 Python 编程语言编写，调用 Python Spark API，通过 `spark-submit` 提交到 Spark 上运行的应用程序。

随机访问

指在一组可寻址元素内的任何给定内存地址访问特定数据项的能力，这与按照磁盘顺序读取数据元素的顺序访问相反。

推荐系统

一个信息系统，目标是预测用户对某些项目的评级或偏好。推荐系统通常使用协同过滤算法实现，它根据类似的用户偏好对项目的整个空间进行过滤。然后使用非负矩阵分解和回归模型等机器学习技术对评级进行预测。

可恢复性

分布式系统的属性，使数据不会在故障发生时丢失。

关系

关系是一组元组，元组的每个元素都是数据域（或数据类型）的成员。在数据库系统中，关系通常是指一个表，行的列都是带类型的。

关系数据库管理系统（**relational database management system，RDBMS**）

根据数据库、表、列、关系的关系型建模原则组织数据的数据库系统。RDBMS 中的查询操作通常使用 SQL 查询语言的某种变体。

水库抽样

一系列随机算法，用于从 n 个项目的列表中随机选择 k 个样本，

其中 n 是非常大的数或未知数。

弹性分布式数据集（**resilient distributed dataset，RDD**）

Spark 中的基本抽象，代表可并行操作的、不可变的、分区的元素集合。

ResourceManager

YARN 中的一个主进程，通过按需给 ApplicationMaster 分配资源（可用的 NodeManager executor 实例），来调度集群上的计算工作。基于预配置策略，ResourceManager 尝试通过容量保证、公平性或服务级别协议来优化集群利用率（让尽可能多的节点肯能忙）。

岭回归

线性回归模型族中的一个正则化模型，通过使用系数的 L2 范数正则化误差最小化函数来惩罚模型复杂性（从而减小模型的偏差）。使用 L2 范数会使权重平滑，降低多重共线性导致的方差的影响。

行键

在 HBase 中，按照唯一的行键对行进行访问和排序。行键本身只是一个字节数组，但是良好的行键设计是在设计健壮的 HBase 数据库数据访问模式时最重要的考虑因素。

可扩展性

分布式系统的属性，使负载增加（更多的数据，更多的计算）将导致性能下降，而不是故障；资源增加，容量也成比例增加。

Secondary NameNode

Secondary NameNode 每隔一段时间复制一次主 NameNode 镜像的编辑日志，定期创建 HDFS 的检查点。它不是主 NameNode 的替代品或备份，而是在重新启动时能让 NameNode 更快恢复。

自适应

一些机器学习模型的属性，可以用新的信息增量更新。数据产品本身应该是自适应的，但若没有增量更新，就得要重新训练模型。

可分的

数据的一种属性，让类可以在特征空间中被超平面分割或分离，具有一些松弛。可分性意味着像支持向量机和随机森林这样的模型将是非常有效的。

序列化

在数据存储上下文中，序列化是一个过程，能将数据结构或对象状态转换为可以存储（例如，存储在文件或内存缓冲器中，或者通过网络连接链路传输），并且之后能够在同一台或别的计算机环境中重新构造回来的格式。

shebang

在脚本起始处的字符序列 `#!`，由字符数字符号和感叹号组成。

单节点设置

在 Hadoop 中，单节点设置在单个机器上安装所有进程（比如 YARN、HDFS、JobHistory Server 等）。也被称为伪分布式设置。

数据槽

数据流中流入数据的接收器或目标。

数据源

可以是数据库、数据存储设备或者进程，发送馈入数据流的输出数据，用于进一步处理或传输到数据槽。

垃圾信息

未经请求和不需要的消息或电子邮件。

Spark Core

构成 Spark 根本的程序内部和抽象（包括 RDD API）的组件、服务和 API。

Spark Python API

Spark 公开的 Python 语言的应用程序编程接口，用于创建 Spark

应用程序。它提供了对 Python RDD 以及 Spark 中许多库工具和代码的访问。

稀疏的

表示数据中有相当高比例的值或单元格不包含实际数据或为“null”。

推断执行

一种能最小化延迟或失败作业的影响的技术。如果检测到慢任务，则立即在相同数据上分配新任务；获胜者是首先完成的任务。

分割

基于某些标准，将数据集划分为多个子集的过程。

分段

将数据传输到中间数据目标或检查点供后续处理的过程。

独立模式

在 Spark 中，此模式可用于在本地计算机的单个进程中运行 Spark。

流数据

不间断或无边界的数据流，稳定且持续地传输和处理。

stripes 和 pairs

请参见“pairs 和 stripes”。

主题专家

主题专家是数据科学家，是数据团队的关键成员。他们为数据问题和模型提供特定领域的知识。另请参见“领域专家”。

有监督的

与无监督机器学习相反，有监督机器学习将模型拟合到数据集，正确答案是提前知道的。分类和回归是有监督机器学习的两个例子。

任务

一个 YARN 作业中的工作单位。在 MapReduce 中，任务是指执行一次 map 操作或 reduce 操作。

任务并行

一种并行形式，在相同或不同的数据集上同时执行多个函数来提升性能。它与数据并行相反，后者是将相同的函数应用于数据集的不同元素。一般来说，map 是数据并行，reduce 是任务并行。

大数据的 **3V**

定义大数据的 3 个属性：容量（volume）、速度（velocity）和多样（variety）。另请参见“容量”“速度”和“多样”。

转换和动作

指两种主要 Spark 操作。转换将 RDD 作为输入，并产生重新格式化的 RDD 作为输出；动作在 RDD 上执行计算，产生一个值返回到 Spark Driver。

元组

一个有限的、不可变的有序元素的集合。

无监督的

与有监督机器学习相反，无监督机器学习基于模式通过实例之间的相似性或距离来拟合模型。这些模型族是无监督的，因为没有“正确的”答案来检验拟合模型的结果或者使误差最小化。聚类是无监督学习的一个例子。

方差

在机器学习中，方差是指给定特定数据点的模型的预测可变性（例如低方差可能表示对于预测的误差量的置信度）。随着方差的减小，偏差将增大。另请参见“偏差”。

多样

数据的结构化格式（CSV、Excel、数据库等）和非结构化格式

（图像、传感器数据、视频等）的范围不断扩大。

速度

　　处理数据的速度或速率。

词汇量

　　一个文本语料库中唯一令牌（或单词）的集合。

容量

　　待处理和存储的数据的量。

worker 节点

　　实现 worker 守护进程的节点，worker 守护进程通常都是 NodeManager 和 DataNode 服务。

工作流管理

　　构建可触发、可参数化、可调度、可自动化的可重复数据处理作业的过程。

数据整理

　　将数据从一种格式（通常为“原始”未处理的格式）转换或映射到另一种格式的过程，使得下游进程可以轻松消费数据以进行分析。

YARN

　　“Yet Another Resource Negotiator”的缩写，是包括 MapReduce 和 Spark 在内的分布式计算引擎的广义集群管理框架。它将处理提交给集群的作业的资源管理和作业调度。

关于作者

Benjamin Bengfort 是一位数据科学家。他虽然住在市里，却对政治（华盛顿的正事）毫无兴趣，反而偏爱技术。他目前正在马里兰大学

攻读博士学位，在那里学习机器学习和分布式计算。他的实验室里确实有机器人（虽然这不是他喜欢的研究领域），但令他懊恼的是，他们老是想给机器人配上刀具或者餐具——大概是为了什么烹饪大奖吧。与其看一个机器人切番茄，Benjamin 更喜欢自己在厨房里研究，他热衷于将法国和圭亚那美食融合，也喜欢各种类型的烧烤。Benjamin 是一名专业编程人员，但他把数据科学家当作自己的事业。他的作品涉及大量主题，从自然语言处理，到使用 Python 的数据科学，再到 Hadoop 和 Spark 分析，应有尽有。

Jenny Kim 是一位经验丰富的大数据工程师。她开发商业软件，也在学术界工作，在海量数据、机器学习以及生产和研究环境的 Hadoop 实施方面有深入研究。Jenny（和 Benjamin Bengfort 一起）建立了一个大型推荐系统，使用网络爬虫收集服装产品的有关信息，并根据交易提供推荐。目前，她正与 Cloudera 的 Hue 团队一道，试图构建能用于 Hadoop 大数据分析的直观界面。

关于封面

《Hadoop 数据分析》的封面动物是牛背鹭（Bubulcus ibis），一种遍布世界的白鹭。起初，牛背鹭只生活在亚洲、非洲、欧洲的部分地区，但在 20 世纪，它已经“入侵了”地球其他大部分区域——经历了一次鸟类最快速、最广泛的自然扩张之后。它主要在热带、亚热带和温带栖息，经常跟随牛或其他大型哺乳动物生活，以这些大型动物驱赶的昆虫或小型脊椎动物为食，因此得名。

牛背鹭是一种白色鸟类。背部、前颈和头部在繁殖期呈橙黄色；与配偶交配之前，喙、腿和虹膜会暂时变成亮红色。非繁殖期的成年牛背鹭几乎通体白羽，黄色喙，灰黄色腿。它身材矮壮，翼展 35 英寸~38 英寸（约为 89 厘米~97 厘米），体高 18 英寸~22 英寸（约为 46 厘米~56 厘米），重约 10 盎司~18 盎司（约为 283 克~510 克）；通常在水边的树木和灌木丛的树枝上筑巢。

因为与牛存在共生关系，牛背鹭深受牧场主的欢迎，被认为是防治牛身上寄生虫的生物手段。但另一方面，当大群牛背鹭在机场的草地边觅食时，可能会造成飞机的安全隐患。此外，它们还可能传播动物疾病，如心水病、传染性法氏囊病甚至新城疫。

O'Reilly 封面上的许多动物都已濒临灭绝，但它们的存在对世界至关

重要。想要了解如何帮助它们，可以登录 animals.oreilly.com。

封面图片来自 *Lydekker's Royal Natural History*。

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

- 微博 @图灵教育：好书、活动每日播报
- 微博 @图灵社区：电子书和好文章的消息
- 微博 @图灵新知：图灵教育的科普小组
- 微信 图灵访谈：ituring_interview，讲述码农精彩人生
- 微信 图灵教育：turingbooks

091507240605ToBeReplacedWithUserId